SELECTA Documentation

Documentation for SELECTA automated workflow engine

Version: 1.0

Authors: Nadim Rahman, Blaise Alako, Peter Harrison and Guy Cochrane

Development of the SELECTA workflow engine was funded by the COMPARE consortium, with funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 643476. This document reflects only the authors views. Neither the European Commission nor its Agency REA are responsible for any use that may be made of the information it contains.

Terminology

Term	Context Definition
Process	A specific fastq file (run) being processed through the SELECTA workflow.
Process ID	An identifier provided to the fastq file which includes the name of the run with the datetime it was downloaded to be processed through SELECTA. Therefore this describes a run - single or paired end reads.
Selection	A rule used to represent a combination of the user, process type (run, study and/or datahub), pipeline name and continuity to be processed through SELECTA.

Introduction to SELECTA

What is SELECTA?

SELECTA is a rule-based workflow engine which runs analytical pipelines, and was initially developed to manage pipelines specifically for the COMPARE community. SELECTA is a combined pipeline scheduler, runner and automatic analysis submitter. This provides automated complete processing of datasets, which consolidates different analysis pipelines into a sole computing and data access environment. SELECTA therefore has application to any project requiring fully automated selection of suitable datasets for processing through analytical pipelines that then automatically load analysis results to an external archive.

How does SELECTA operate?

Configuration files and data processing definitions enable set criteria to be implemented in order to automatically acquire datahub (group) reads for processing from the European Nucleotide Archive (ENA), allowing for different complexities of data grouping. In essence SELECTA directs analysis pipelines to run at the datahub level, which includes various ENA projects with runs associated to these projects. However the flexibility of SELECTA also enables for data processing to occur at the ENA study level (including all runs within the study) or at the level of just a single run. Upon acquiring the relevant data, SELECTA's user-defined rule definition configuration automatically instructs the processing and analysis of the reads, with rules that instruct which pipelines to run on what datahubs. Finally, automatic submission of analysis results back to the ENA mitigates the burden for a submitter when handling bulk submissions and ensures analysis results are consistently and accurately recorded as analysis metadata and file descriptions are generated by SELECTA itself.

The SELECTA Codebase

The SELECTA codebase is available from GitHub under an Apache 2.0 license: https://github.com/EBI-COMMUNITY/ebi-selecta.

This GitHub repository contains the codebase for the workflow engine only, any individual managed pipelines for use by SELECTA need to be installed separately. For COMPARE these can be found publicly elsewhere (see SELECTA Setup and Managed Pipelines). Ensure that SELECTA has been instructed on where to locate the pipelines to be operated (see Integrating Pipelines).

The SELECTA codebase includes instructions on installation, focusing on a portable and simple installation and setup through the usage of Docker (https://www.docker.com/). Additionally, the SELECTA Application Programming Interface (API) is also set up, with documentation found at the following site: https://github.com/EBI-COMMUNITY/ebi-selecta/blob/master/service_api/README.md. Installation has been detailed below.

SELECTA Docker Installation

The SELECTA toolkit includes a set of Dockerfiles within the GitHub repository (in the prefixed ‘service_’ directories). Additionally, configuration docker-compose files are required with this to launch and run the SELECTA framework without installing individual components of SELECTA.

The repository also contains an init_swarm.sh script for docker swarm environment initialisation and a stack file for launching the framework in a docker swarm cluster. Additionally, a script - rm_swarm.sh - is also included to remove a created swarm cluster environment.

Install Docker

SELECTA requires installation of Docker and Docker-compose

Ubuntu:

1. Add GPG key for official Docker repository to the operating system.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
```

2. Add Docker repository to the APT (Advanced Package Tool) sources.

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

3. Update the Ubuntu package database with the newly added docker packages repository.

```
sudo apt-get update
```

4. Install Docker Community Edition.

```
sudo apt-get install -y docker-ce
```

5. Verify that the Docker daemon is running.

```
sudo systemctl status docker
```

6. Install docker-compose and ensure it is executable.

```
sudo curl -L
https://github.com/docker/compose/releases/download/1.18.0/docker-compos
e-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose && sudo chmod
```

```
+x /usr/local/bin/docker-compose
```

7. Verify that the installation of docker-compose is successful.

```
docker-compose --version
```

Windows/Mac:

1. Install the executable from the docker-compose page - https://docs.docker.com/compose/install/

Obtain Codebase(s)

SELECTA

Clone the repository above for SELECTA code.

```
git clone https://github.com/EBI-COMMUNITY/ebi-selecta
```

Pipelines

If utilising for COMPARE, the codebase of the individual pipelines should be obtained. Installation instructions for the pipelines can be found at the relevant websites (see Integrated Pipelines).

Alternatively, obtain the codebase of the desired analysis pipeline for integration into SELECTA.

Configuration files to run SELECTA

Now that you have installed SELECTA and one or more pipelines either from COMPARE or from other sources, you need to complete the configuration files to finalise the SELECTA setup.

properties.txt

If non-Dockerised SELECTA is utilised, a `resources/properties.txt` file is required for configuration. Ensure that the paths/values for each of the parameters within the file are valid. See 'Properties File' in Appendix 1 for more information on the definition of each parameter.

docker-compose

However if using SELECTA with Docker, the docker-compose YAML file coupled with the `resources/properties.txt` file also is required for configuration. Note the volume mount paths, for which local paths correspond to those on the container.

SELECTA API

SELECTA has an Application Programming Interface (API) used for various management services. For example it can be used to list analyses, obtain status of a particular run, lookup run details and return information on a study and datahub. Additionally, the API can be used to register new data to be processed and restart a run status in order for re-processing. The endpoints and usage of the API have been outlined below. SELECTA-API documentation is found here: https://github.com/EBI-COMMUNITY/ebi-selecta/blob/master/service_api/README.md.

Using SELECTA

This section describes the usage of SELECTA once set up.

Initiating SELECTA Workflow

Once SELECTA is set up, the following steps will initiate the SELECTA workflow on a new selection. This selection can be an individual sequence read, a project - group of reads, or a datahub - one or more projects each with their own sets of sequence reads.

1. **Create a datahub account**

This account is required to associate uploaded data and associated metadata. Therefore, runs and experiments form the basis of samples which are housed within projects in the datahub. Note: SELECTA can handle single or paired-end reads.

To create a datahub account, send a request to datasubs@ebi.ac.uk with a short description of the project/collaboration.

2. **Store datahub account details in 'account' table**

These credentials are retrieved by SELECTA to access the sequence read(s) and their associated metadata for processing.

Example:
The following command generates an account for dcc_test using SELECTA's API.

```
curl -d '{"account_id":"dcc_test","account_type":"datahub","email":
"selecta@ebi.ac.uk","password": "letmein"}' -H "Content-Type:
application/json" -X POST http://localhost:5002/account
```

Note: A similar SQL command can be carried out to directly include this information into the 'account' table.

3. **Insert a row into 'process_selection' table**

Also known as 'Input to Selection'. This new row presents the selection which requires processing. Ensure that the selection_to_attribute_start, selection_to_attribute_end and selection_to_attribute_error fields are empty (set to NULL).

Example:
(Note only provide study and/or run information if this is known).
The following command generates a rule in the 'process_selection' table for processing data from dcc_dvorak datahub through UAntwerp_Bacpipe. By setting 'Continuity' to 'YES', it is ensured that the analysis of the run is an ongoing process.

```
curl -d '{"datahub": "dcc_dvorak","run_accession":
"ERR1102130","pipeline_name": "UAntwerp_Bacpipe","public": "NO","webin":
"Webin-45433","continuity": "YES","process_type": "run"}' -H
"Content-Type: application/json" -X POST
http://localhost:5002/input2selection
```

Check Processing Status

The following can be carried out using SELECTA's API (see SELECTA API), as opposed to direct interaction withe the database. To check whether runs of a selection have been processed through the SELECTA workflow, check the process_report table. Both of the following are
Example:

```
SELECT process_report_start_time, process_report_end_time FROM
process_report WHERE selection_id = X;
```

Similarly, a specific process_id (run) can be checked from the process_report table.
Example:

```
SELECT process_report_start_time, process_report_end_time FROM
process_report WHERE process_id = 'XXXXXXX';
```

The returned values will indicate whether the process_id(s) have started and/or completed the SELECTA workflow.

Check SELECTA Stage Processing

As above, the following can be carried out using SELECTA's API (see SELECTA API). To check the processing status for a selection through a specific SELECTA workflow stage, check the process_stages table for a given selection_id. Alternatively, view the status for a process_id by specifying a process_id in the select statement.
Example:

```
SELECT * FROM process_stages WHERE selection_id = X;
```

The stage_start and stage_end fields provide timestamps for commencing and completing an individual stage (presented in the stage_name field). Furthermore, the stage_error field provides error messages for processing at a particular stage_name for a selection/process_id.

Manual SELECTA Usage

If running SELECTA manually, run the following scripts sequentially (as also detailed in Appendix 2): selection_to_attribute.py, data_provider.py, core_executor.py, analysis_reporter.py and process_archival.py. Each file requires the SELECTA configuration file (properties.txt) as an argument.

selection_to_attribute

```
docker run -ti --rm --network=ebi-selecta_postgres
embl-ebi/selecta_selection_to_attribute:1.0
/usr/scr/app/scripts/selection_to_attribute.py -p
/usr/scr/app/resources/properties.txt
```

data_provider

```
docker run -ti --rm --network=ebi-selecta_postgres
embl-ebi/selecta_data_provider:1.0 /usr/scr/app/scripts/data_provider.py
-p /usr/scr/app/resources/properties.txt
```

core_executor

```
docker run -ti --rm --network=ebi-selecta_postgres
embl-ebi/selecta_core_executor:1.0 /usr/scr/app/scripts/core_executor.py
-p /usr/scr/app/resources/properties.txt
```

analysis_reporter

```
docker run -ti --rm --network=ebi-selecta_postgres
embl-ebi/selecta_analysis_reporter:1.0
/usr/scr/app/submission/analysis_reporter.py -p
/usr/scr/app/resources/properties.txt
```

process_archival

```
docker run -ti --rm --network=ebi-selecta_postgres
embl-ebi/selecta_process_archival:1.0
/usr/scr/app/scripts/process_archival.py -p
/usr/scr/app/resources/properties.txt
```

For a more automated version of this, generate periodic individual cronjobs for each stage.

SELECTA Docker Swarm

Initialise SELECTA Swarm Cluster
Run the command below to initialise:

```
./init_swarm.sh
```

To confirm the cluster has been created:

```
docker node ls
docker node inspect node-id
```

Launch SELECTA in Swarm Cluster
Run the following command to launch:

```
docker stack deploy -c docker-stack-compose.yml selecta
```

This command pulls the required SELECTA images from docker hub and launches all SELECTA services as configured within the YAML file.

The SELECTA framework integrates a swarm visualiser, accessible at http:://localhost:8080.

To scale up core_executor service, run the command below:

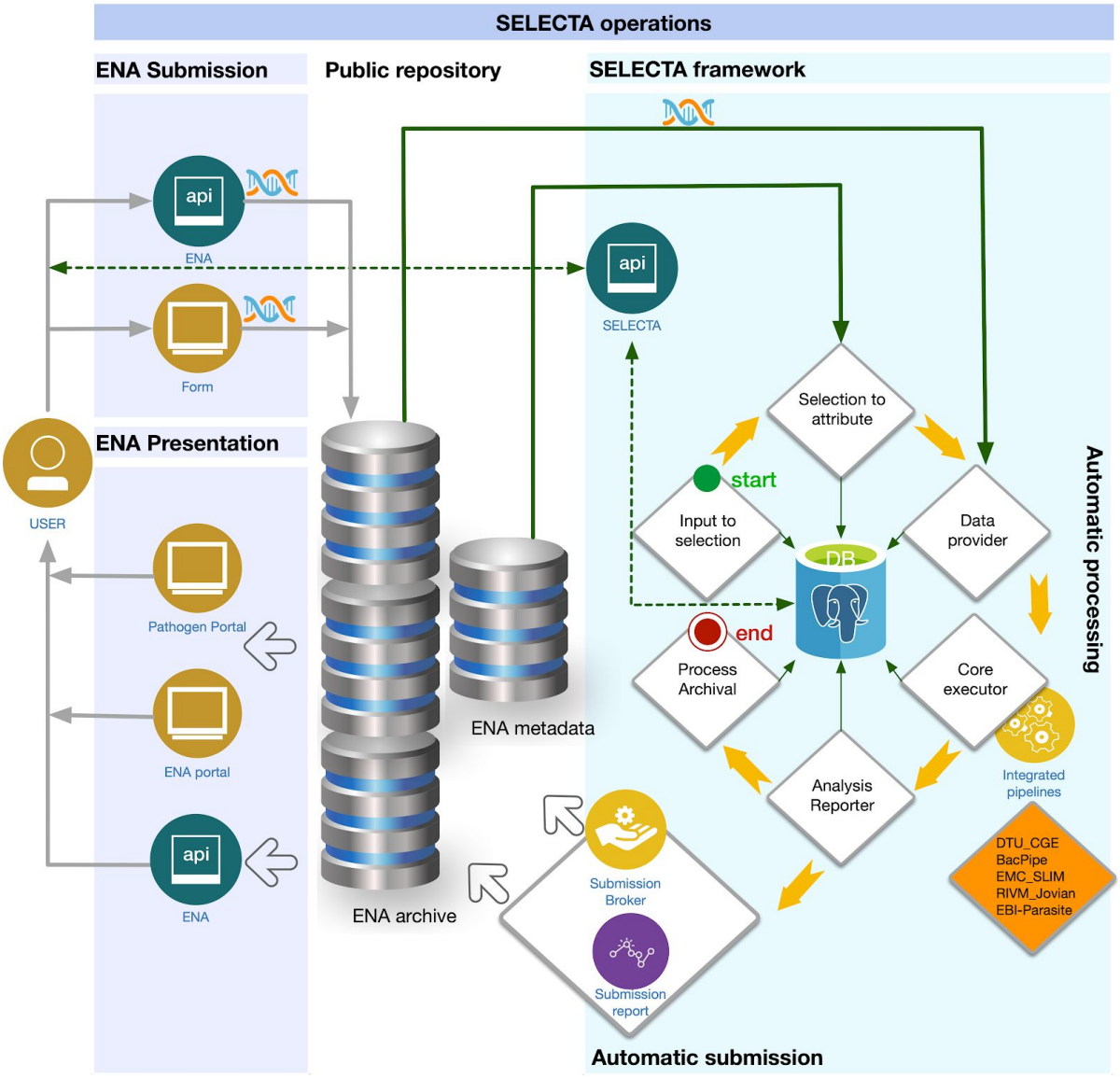
```
docker service scale selecta_core_executor=10
```

Ensure that the service has scaled up via the swarm visualiser at http:://localhost:8080.

Remove Swarm Cluster
Run the following command to remove the swarm cluster:

```
./rm_swarm.sh
```

Detailed Overview of Software Architecture



The diagram above presents SELECTA's software architecture in detail. Currently, sequence files are obtained from storage/public repository (ENA archive), submitted through a submission form or API. More information on submission can be found in the following publication - https://www.biorxiv.org/content/10.1101/555938v1. Specific file locations are defined in datahub reporting during ENA submission. This is among other metadata obtained during submission, stored and retrieved in ENA metadata and is required for processing and analysis annotation.

At each major stage of the SELECTA framework, references and alterations to the SELECTA PostgreSQL database take place (shown in the SELECTA framework above and discussed in more detail in Appendix 2). The database can be accessed to obtain processing information for a selection - as mentioned in Using SELECTA, below in Troubleshooting or using the SELECTA API. Pipeline processing occurs during core

execution, where integrated pipelines are run according to the metadata provided in datahub reporting. Note that, at this stage pipeline databases may be required - maintained externally to SELECTA.

During analysis reporting, automatic submission is carried out where SELECTA brokers the submission of the pipeline processing analysis, generating a submission and analysis XML for each run processed. This enables for annotated data along with the actual analysis to be presented back to the user (as shown in ENA presentation). Specifics about the integration of pipelines and expected inputs and outputs can be found in ‘Integrating Pipelines’.

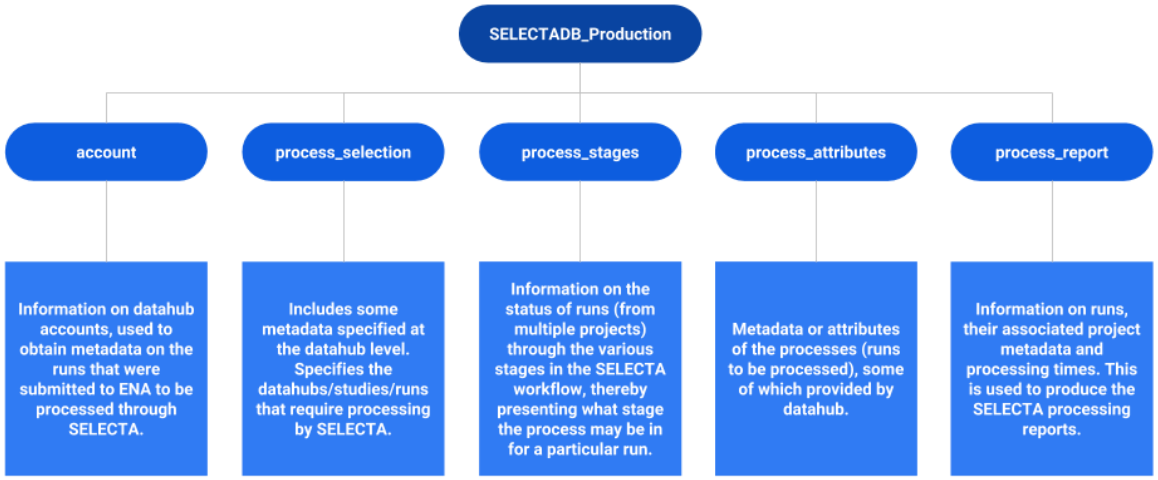
SELECTA Database

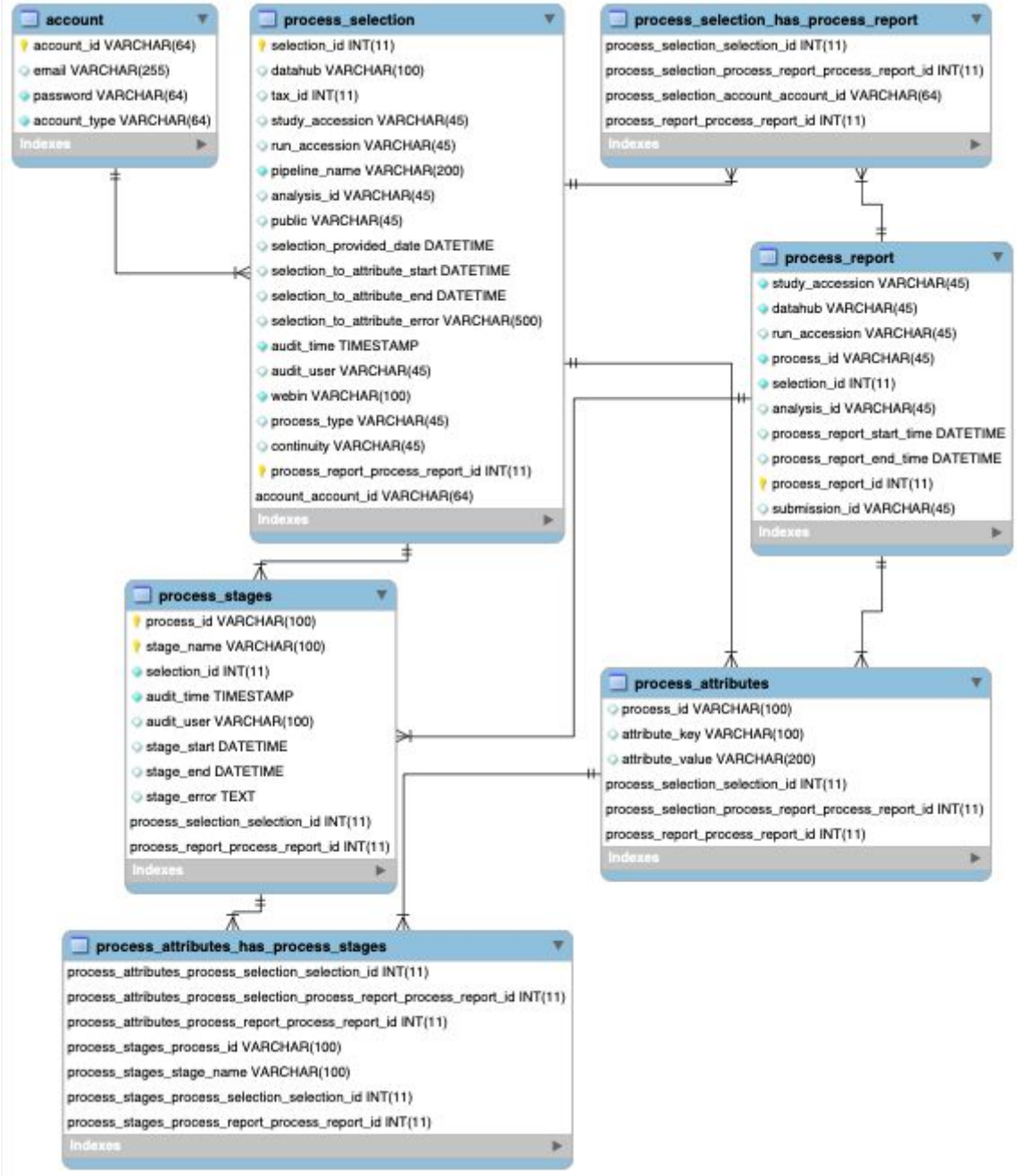
Background

SELECTA’s PostgreSQL database (https://www.postgresql.org/) plays a major role in managing the processing of and tracking runs. SELECTA frequently retrieves or updates information from the table depending on what process is running. The tables with their descriptions have been presented in the diagram below, with a more detailed schema presented below this. Separate development and production instances of the database enable for testing and development.

Database Schema

Currently, SELECTA utilises a database consisting of five main tables. The database is altered or updated as part of SELECTA run processing. Users will provide metadata and information on data submitted. This is retrieved through the user-specified datahub name, study, or run accession by SELECTA through ENA-API which is stored in the database as and when required.




account
account_id VARCHAR(64)
email VARCHAR(255)
password VARCHAR(64)
account_type VARCHAR(64)
Indexes
process_selection
selection_id INT(11)
datahub VARCHAR(100)
tax_id INT(11)
study_accession VARCHAR(45)
run_accession VARCHAR(45)
pipeline_name VARCHAR(200)
analysis_id VARCHAR(45)
public VARCHAR(45)
selection_provided_date DATETIME
selection_to_attribute_start DATETIME
selection_to_attribute_end DATETIME
selection_to_attribute_error VARCHAR(500)
audit_time TIMESTAMP
audit_user VARCHAR(45)
webin VARCHAR(100)
process_type VARCHAR(45)
continuity VARCHAR(45)
process_report_process_report_id INT(11)
account_account_id VARCHAR(64)
Indexes
process_selection_has_process_report
process_selection_selection_id INT(11)
process_selection_process_report_process_report_id INT(11)
process_selection_account_account_id VARCHAR(64)
process_report_process_report_id INT(11)
Indexes
process_report
study_accession VARCHAR(45)
datahub VARCHAR(45)
run_accession VARCHAR(45)
process_id VARCHAR(45)
selection_id INT(11)
analysis_id VARCHAR(45)
process_report_start_time DATETIME
process_report_end_time DATETIME
process_report_id INT(11)
submission_id VARCHAR(45)
Indexes
process_stages
process_id VARCHAR(100)
stage_name VARCHAR(100)
selection_id INT(11)
audit_time TIMESTAMP
audit_user VARCHAR(100)
stage_start DATETIME
stage_end DATETIME
stage_error TEXT
process_selection_selection_id INT(11)
process_report_process_report_id INT(11)
Indexes
process_attributes
process_id VARCHAR(100)
attribute_key VARCHAR(100)
attribute_value VARCHAR(200)
process_selection_selection_id INT(11)
process_selection_process_report_process_report_id INT(11)
process_report_process_report_id INT(11)
Indexes
process_attributes_has_process_stages
process_attributes_process_selection_selection_id INT(11)
process_attributes_process_selection_process_report_process_report_id INT(11)
process_attributes_process_report_process_report_id INT(11)
process_stages_process_id VARCHAR(100)
process_stages_stage_name VARCHAR(100)
process_stages_process_selection_selection_id INT(11)
process_stages_process_report_process_report_id INT(11)
Indexes

Managed Pipelines

Integrated Pipelines

The core execution stage of SELECTA processing includes running integrated pipelines developed by members of the COMPARE community. The following pipelines are fully integrated and functional in the system. Firstly, the Erasmus Medical Center Virus analysis pipeline (EMC_SLIM), the Center for Genomic Epidemiology bacterial genome analysis pipeline of the Danish Technical University (DTU_CGE), the University of Antwerp bacterial genome analysis pipeline (UAntwerp_Bacpipe) and the National Institute for Public Health and the Environment metagenomic analysis pipeline (RIVM_Jovian). These pipelines have been integrated through extensive collaboration with the specific groups. Additional or alternative pipelines can be integrated with relative ease within SELECTA through collaboration. This may require tweaks in pipeline code.

EMC_SLIM

Developers: The Erasmus Medical Center (EMC), Netherlands

Links: https://github.com/EBI-COMMUNITY/slim_emc

Overview:
SLIM includes several bioinformatics tools for the de novo assembly of Illumina HiSeq paired-end or Ion Torrent single end reads to contigs and contigs classifications. This also includes an initial generic quality control of short read sequences through the removal of common adapters and low quality reads. SPAdes is utilised on the set of reads passing quality control. Generic parameters are utilised by default providing a reasonable de novo assembly performance across a broad range of sequence data types. This provides a useful starting point for analysis. SLIM classifies the contigs based on a translation of all six reading frames of the contig and a usearch screen for homologies to viral proteins. The main output of SLIM classification includes a single tab-separated value table listing each contig and showing protein homology above 30% to sequences in the virus family-based database. As with the pipelines below, intermediate files are provided within a compressed directory.

Embedded Command:
A python2 script is called to run the SLIM pipeline, with the following arguments: first fastq file, second fastq file (if paired reads), run accession ID, SLIM property file and working directory.

Additional Dependencies:
Further dependencies have been specified within the SLIM property file. This includes the .jar file for QuasR, USEARCH sequencing analysis tool, SPAdes genome assembly tool, viral database list file, Illumina core, BioPython, TeilenQ for paired, and single end reads.

DTU_CGE

Developers: Technical University of Denmark (DTU)

Publication: Thomsen M.C.F., Ahrenfeldt J., Cisneros J.L.B., Jurtz V., Larsen M.V., Hasman H., Aarestrup F.M. and Lund O., (2016). A Bacterial Analysis Platform: An Integrated System for Analysing Bacterial Whole Genome Sequencing Data for Clinical Diagnostics and Surveillance. *PLoS: ONE*, 11(6), e0157718. (DOI:10.1371/journal.pone.0157718).

Links:
Source code - https://bitbucket.org/genomicepidemiology/cge-tools-docker/src/master/
Docker - https://bitbucket.org/genomicepidemiology/cge-tools-docker

Overview:
Bacterial whole-genome sequencing (WGS) has become ubiquitous in microbiological laboratories due to the low cost of sequencing. However, a major obstacle for applying sequencing technologies in a clinical setting is the availability of transparent and automatic bioinformatics tools. DTU_CGE consolidates several published state of the art web-based tools for the analysis of bacterial genomes. The pipeline involves automatically identifying the bacterial species from multiple bacterial isolates. If necessary, the genome is assembled to identify multi-locus sequence type, virulence genes and anti-microbial resistance genes. During mid-2018, the CgMLST and Salmonella type finder capabilities were included for the Euopean Centre for Disease Prevention and Control (ECDC) trial. The analysis output includes a summary report of the analysis enabling for rapid overview of the specific processes. The DTU_CGE pipeline demonstrated reliability during benchmarking using datasets previously used to test the individual processes. The pipeline correctly predicts species and the majority of expected genes automatically and provides a simple automated analysis interface for bacterial WGS data. Moreover, the platform successfully exploits WGS in assisting clinical diagnostics and pathogen surveillance.

Embedded Command:
SELECTA runs the Singularity-containerised CGE pipeline with two binding points (CGE databases folder and working directory). The BAP.py script is called to initiate the workflow, with the following arguments: first fastq file, second fastq file (if paired reads) and sequencing platform.

UAntwerp_Bacpipe

Developers: The University of Antwerp, Belgium

Links:
BacPipe Software - https://github.com/wholeGenomeSequencingAnalysisPipeline/BacPipe/releases
Docker Image - https://github.com/basilbritto/bacpipev1 (To pull docker image: mahmed/bacpipe)

Overview:
Adopting whole genome sequencing (WGS) in a microbiological clinical diagnostics and infection management requires knowledge of complicated bioinformatics tools. To overcome this, BacPipe was developed for bacterial genome analysis. It is an amalgamation of several open-access bioinformatics tools, organised in a logical order for bacterial WGS. The workflow commences with a quality checking of reads, followed by genome assembly, annotation and identification of resistance and virulence genes. Furthermore, BacPipe can simultaneously analyse several strains of bacteria in order to gain understanding of the evolutionary relationship and derive a bacterial transmission route. Using the Methicillin-Resistant Staphylococcus Aureus (MRSA) outbreak WGS dataset amongst other datasets, BacPipe was validated. The results of the analysis are consolidated into a single MS Excel worksheet for clinical overview and interpretation. Therefore BacPipe is highly beneficial in any clinical setting due to the relatively rapid analysis and interpretation of pathogen sequence dataset. Consequently, this eases its application into routine patient care within the hospital and monitoring infection control in public health.

Embedded Command:
A python2 BacPipe program script is called to run the pipeline with the following arguments: YAML configuration file, OS and number of processors.

Additional Dependencies:
Aside from the BacPipe python script, BacPipe requires Prokka (from Bioconda) which has been specified within the properties file.

RIVM_Jovian

Developers: National Institute for Public Health and the Environment, Netherlands

Links: https://github.com/DennisSchmitz/Jovian (private, request for access)

Overview:
Jovian is a metagenomics analysis workflow designed for wet-lab scientists and clinicians. Jovian consolidates established open-source bioinformatic analytical tools in one easy to use and transparent environment. Its analytical workflows automatically process and analyze Illumina sequence reads from human clinical samples into clinically actionable information. The sequence reads first undergo a quality control followed by human sequence data removal to enforce patient privacy. The pipeline assembles the sequence reads into scaffolds and possible full viral genomes. Assembled sequences are classified up to species level and viral sequences taxonomically labeled at the sub-species level.
Jovian aims at improving classical molecular diagnostic and pathogen surveillance capacity. Jovian provides the clinician or researcher with a taxonomic classification of the microbial and viral species detected in clinical samples via an interactive web-report and several Jupyter notebook visualization tools for analysis results interpretation. Additionally, Jovian enforces an audit trace; this is especially beneficial for clinical analysis reproducibility and logging.

Embedded Command:

Additional Dependencies:
Snakemake, bioconda

Adaptations to Managed Pipelines for Integration

To integrate the COMPARE community pipelines above into SELECTA, some pipelines require adaptations detailed in this section.

DTU_CGE

The docker image (see Managed Pipelines - DTU_CGE) requires porting into Singularity. This was carried out to avoid security issues surrounding container images and mapping to users.

UAntwerp_BacPipe

The docker image was not utilised as this is graphical user interface (GUI) -based. Follow ‘Local Installation’ described in the BacPipe GitHub repository.

RIVM_Jovian

The pipeline requires an ‘initialisation’ stage whereby the SELECTA process directory for a given process_id must include the pipeline profile, Jovian scripts, pipeline environment setup files, Snakefile, bin folder with pipeline micro-services and .git file. Furthermore, a fastq directory is required for storing the fastq file(s).

Integrating Pipelines

SELECTA’s modularity is a major benefit for integrating and testing additional pipelines with relative ease. This section outlines the steps for integrating pipelines to run within the SELECTA framework.

Required Pipeline Input and Output

Integrated pipelines require one or a pair of genomic sequencing files (e.g. fastq) as input. These files are obtained in .gz format during data provision before being processed during core execution.

Integrated pipelines must produce at least one summary TSV file (ASCII format) presenting the results of the pipeline processing. A major advantage of utilising ASCII format, includes

ease in discoverability after indexing by the ENA infrastructure, following which is presented at the ENA web browser (www.ebi.ac.uk/ena) and Pathogens Portal (www.ebi.ac.uk/ena/pathogens). This, along with a tar-gzipped process directory are submitted to the ENA as analysis to be presented downstream. Therefore, if the pipeline does not explicitly produce a summary TSV file, the pipeline requires adaptation. If the pipeline does not tar-gzip the processing directory, then a function which does this is required in scripts/pipelines.py (see Steps for Pipeline Integration).

Overview of Steps for Pipeline Integration

The major steps required for pipeline integration have been outlined below:

1. **Create a new class object for new pipeline in pipelines.py.**
2. **Create core executor function for the pipeline in core_executor.py.**
3. **Ensure any dependencies (if any) are met.**
4. **Include any pipeline-specific requirements in properties files(s).**
5. **Alter the properties class object according to step 4.**

Class Object

An example template for pipeline class:

```
class PipelineName:

def __init__(self, fq1, fq2, ...):
    """
    Initialize object variables
    """

def command_builder(self):
    """
    Command builder for tool/pipeline to be utilised
    :return: A command which is run to execute the pipeline processing on input
files
    """
    # Output must meet requirements - See Integrating Pipelines: Required
Pipeline Input and Output

def command_builder_mock(self):
    """
    Mock command builder for tool/pipeline to be utilised
    :return: A mock command for the purpose of quick testing
    """
    # Output must meet requirements - See Integrating Pipelines: Required
Pipeline Input and Output

def run(self, command):
    """
```

```
    Run the command constructed by the command builder method
    :param command: COMPARE pipeline-tool command built in command builder method
    :return: Job ID is LSF used, otherwise none
    """
    # Utilise LSF functionality if available

"""
Some potential post-processing steps
"""
def copy_src_into_dest(self, src, dest):
    """
    Copy pipeline intermediate files into a destination directory for the purpose
of archiving
    :param src: Source directory
    :param dest: Destination directory
    :return: None
    """

@staticmethod
def delete_empty_files(folder):
    """
    Delete empty files from a directory
    :param folder: Directory containing empty files
    :return: None
    """

@staticmethod
def make_tar_gzip(src, des):
    """
    Create an archived and compressed directory
    :param src: Directory to be archived
    :param des: Name of the archive and compressed directory
    :return: None
    """

@staticmethod
def zip_dir(src):
    """
    Compress a directory with zip tool
    :param src: Directory to be compressed
    :return: None
    """

@staticmethod
def del_file(filename):
    """
    Delete a single file
    :param filename: File to delete
    :return: None
    """
```

```
def post_process(self):
    """
    Consolidate the analysis into a main folder to be archived and submitted to
the ENA
    :return: None
    """
    # This function may call on static methods defined above

def execute(self):
    """
    Execute pipeline calling on command builder and run methods defined above
    """
```

Within scripts/pipelines.py, include a new class object for the pipeline to be integrated. A template has been shown above. Initialise this with the relevant attributes required to run the pipeline from the SELECTA information. These should include properties defined in the resources/properties.txt file coupled with metadata obtained during the SELECTA workflow (e.g. fastq locations and accessions).

Execution Using the Class Object

The class object generated above should now contain all the attributes and processing that the integrated pipeline requires. However it has not been fully embedded into the SELECTA framework.

Create a new function within scripts/core_executor.py which utilises the class object to run the pipeline (e.g. execute_dtu_cge). Therefore all the attributes used to initialise the class object will be arguments passed to it within this execution function.

Remember to add a condition within the execute function of scripts/core_executor.py which calls on the function created above to run when called upon.

Meet Dependencies

Ensure that any dependencies for the pipeline are met. A containerised pipeline is likely to have much fewer if no dependencies aside from the containerisation software. Additionally, this future-proofs the pipeline to run despite updates to dependencies.

Dependencies in the case of the integrated pipeline also includes any databases required for its functioning. These databases should be maintained outside of SELECTA.

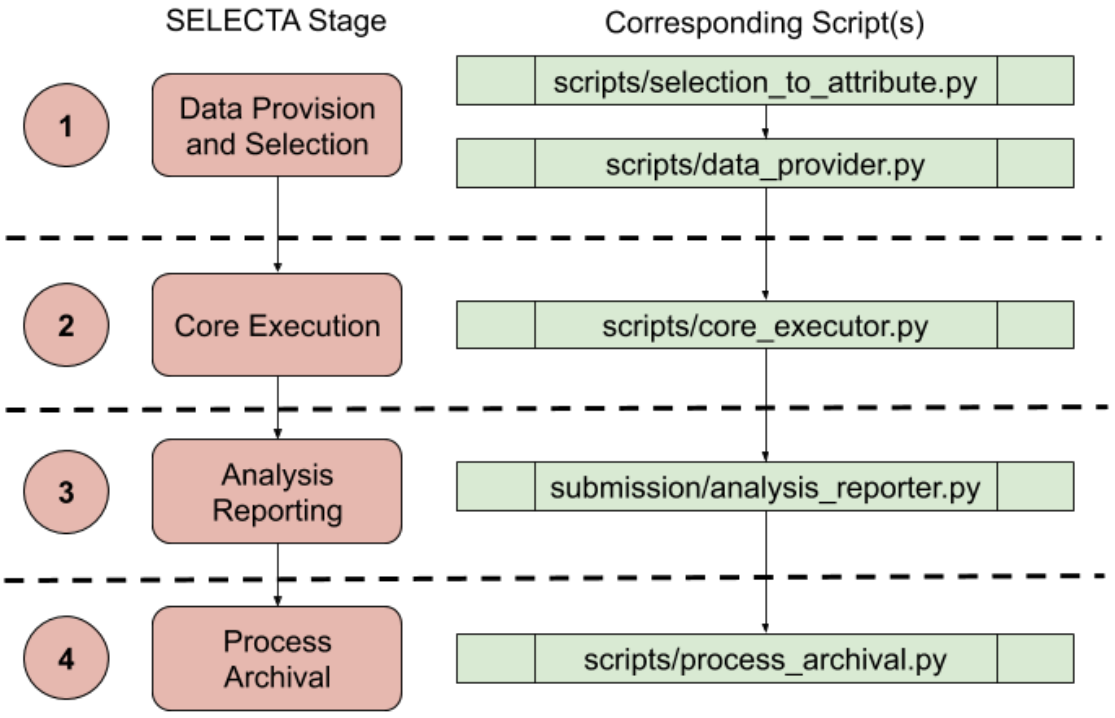
Reading Dependencies from Properties

Add the dependencies to SELECTA's properties file. These dependencies should only include those utilised with the SELECTA codebase (e.g. location of database, required to pass in as an argument for the pipeline to run). The parameter name and value must be

separated by a colon (:). See Appendix 1 - Properties File for more information on the properties file.

Any changes made to the properties file will not be picked up by SELECTA unless included as part of the properties class object within selectadb.py. Alter this according to the properties file and ensure that the properties have been addressed in the scripts/core_executor.py functions accordingly.

SELECTA Stage Components



SELECTA has been developed in a highly modular format, including a chain of individual stages (shown above) acting as separate micro-services. Each of the stages can be run automatically or manually, through the python scripts, detailed in this section. The scripts require the properties file as an argument:

```
[Arguments]
      -p (--properties_file)  File-path to properties file in resources
directory of codebase.
```

Selection to Attribute

scripts/selection_to_attribute.py
Initiates SELECTA workflow, given a selection in the process_selection table with a NULL start date and a NULL selection errors, fetches all known metadata for a run through ENA repository to populate the process_stages, process_attributes and process_report database tables with run metadata. This enables for appropriate annotation of analysis later, when required.

Data Selection and Provision

scripts/data_provider.py

For a list of process_id in process_stages table yet to undergo data provision, downloads the corresponding fastq(s) from the ENA FTP repository. This is the primary data consumed by integrated analysis workflows.

Core Execution

scripts/core_executor.py
For a list of process_id in process_stages table yet to undergo core execution (having successfully completed data provision), call on the appropriate pipeline as defined in the configuration rules, constructs the necessary command in the backend to process run the sequencing data obtained in data provision.

Analysis Reporting

submission/analysis_reporter.py
For a list of process_id in process_stages table yet to undergo analysis reporting (having successfully completed data provision and core execution), create XML objects - analysis.xml and submission.xml defining the analysis and then submission, respectively, to ENA. Documentation for creation of these XML objects can be found here: https://ena-docs.readthedocs.io/en/latest/submit/general-guide/programmatic.html. This submits the pipeline analysis results back to the ENA whilst providing submission and analysis identifiers for subsequent discovery of data within the ENA or Pathogens Portal.

Process Archival

scripts/process_archival.py
For a list of process_id in process_stages table yet to undergo process archival (having completed data provision, core execution and analysis reporting), archive the submitted analysis results into a dedicated ENA archive. Additionally it performs some sanity checks.

Log Files

SELECTA produces two types of log files, stage-specific and LSF specific.

Stage-Specific

The first type of log file produced includes the stage-specific log file (e.g. core_executor.log). The contents of these log files differ according to which SELECTA stage/script is run, however on the whole each script presents information on what SELECTA is doing at any given point within each stage. These log files may also present some errors which may have occurred during the workflow. Stage-specific log files are generated from the cronjobs as when each C-shell script for a given SELECTA stage is run (which then runs the corresponding python script), the command line output is saved within that specific SELECTA stage log file.

LSF-Specific

The second form of log file is produced if LSF capabilities exist and are utilised. These are specifically output and error files for a given process ID (run) when submitted as an LSF job.

Therefore these are separate log files reporting on the progress of the submitted LSF job for the particular SELECTA stage within the workflow. The stages which require LSF jobs and hence produce these types of log files include selection_to_attribute (to download datahub metadata), data_provider (to download the submitted sequence files - fastq) and core_executor (to run the pipeline on the read sequence file). These files are named in the following format: [SELECTA process name]_[process ID].[LSF job ID]. Note that the process ID, includes the run name and the time and date which the run was first processed within SELECTA.

Troubleshooting

Identifying Errors

SELECTA has been generated to aid as much as possible when troubleshooting. Firstly, errors are recorded within the error field of SELECTA database tables.

Secondly, log files can be used to trace errors due to the specific process IDs provided to runs. The stage-specific log files should present with any errors that occurred when running the stage-specific python script (during the processing). For SELECTA stages which run using LSF, the stage-specific logs can be used in conjunction with the LSF-specific log files for further investigation into errors. (See Log Files for more information).

Lastly, browsing the process/archive directory for the specific failed process can also aid in troubleshooting. These directories have been named according to the process_id, which benefits traceability.

Once the error has been identified and addressed, the SELECTA workflow can be restarted for a particular run, see Restart Processing below.

Selection to Attribute

Database Check

Progress of the processing of a selection can be checked by querying the corresponding selection_id in the process_selection table of SELECTA's database:

```
SELECT * FROM process_selection WHERE selection_id=X;
```

The fields of note are selection_to_attribute_start, selection_to_attribute_end and selection_to_attribute_error. These fields can be searched for explicitly as opposed to searching for all fields.

Log File Check

Coupled with the database check, the selection_to_attribute.log file can be searched for a progress check.

Attributes Check

Successful processing of a selection at this stage should result in the introduction of attributes for the selection_id within the process_attributes table. This can be checked using the following query in the process_attributes table:

```
SELECT * FROM process_attributes WHERE process_id IN (SELECT process_id
FROM process_attributes WHERE attribute_key='selection_id' AND
attribute_value=X);
```

Here all the process_ids for a selection are obtained by querying the attribute_key and attribute_value fields with the selection_id. From these process_ids, all the attributes are obtained, providing all attributes for each process_id for a given selection_id.

Data Provision

At some point within the following checks, it may be necessary to identify the status of the selection processing during the selection to attribute stage (see above).

Database Check

The progress of a selection through this stage can be checked through the process_stages table. A list of all process IDs that have failed data provider processing can be obtained:

```
SELECT * FROM process_stages WHERE stage_name='data_provider' AND
stage_start IS NOT NULL AND stage_error IS NOT NULL;
```

Here, a list would be returned, presenting information of process IDs which have started the data provision stage, however presented with an error. The stage_end field may also be set to NULL (which can be added to the query above).

The process_ids that have failed processing can be searched individually, through the query below. The fields of note include stage_start, stage_end and stage_error:

```
SELECT * FROM process_stages WHERE stage_name='data_provider'  AND
process_id='XXXXXXXX';
```

The stage_end field does not contain a timestamp, unless processing has completed without error.

Alternatively the progress of a specific selection can be searched using the following query.

```
SELECT * FROM process_stages WHERE stage_name='data_provider' AND
selection_id=X;
```

Log File Check

The data_provider.log file can be searched for the process_id for information on its processing. If LSF has been utilised, then job IDs will be present which can be used to trace processing information within the corresponding LSF-specific log file.

Processing Directory Check

It is highly recommended a check of the processing directory is carried out as part of troubleshooting. A successful run of this stage for a process ID should result in the fastq file(s) being present in the corresponding process directory (named after the process_id).

Core Execution

A check on the status of the processing of the process ID during the data provision stage should be carried out (see above).

Database Check

Obtain information on the process IDs which have failed the core execution stage using the query below. This queries the process_stages table for rows which present with errors in the stage_error field.

```
SELECT * FROM process_stages WHERE stage_name='core_executor' AND
stage_start IS NOT NULL AND stage_error IS NOT NULL;
```

The process_ids can be queried individually using the following query:

```
SELECT * FROM process_stages WHERE stage_name='core_executor’ AND
process_id='XXXXXXXXXX';
```

If an error has occurred, the stage_error field should contain the error message. The checks mentioned below should also be carried out.

Log File Check

The progress of a selection or process ID can be searched for within the core_executor.log file. This contains information on steps carried out and also if LSF was utilised, job IDs.

These job IDs can also be used to check the appropriate LSF-specific log file should LSF be utilised.

Processing Directory Check

The process directory for a given process_id provides clues to the progress during core execution. As this stage involves integrated pipeline execution and processing, the contents of the directory should include a summary TSV file and a sub-directory of the pipeline processing that is also tar-gzipped. The contents of the sub-directory differ depending on the actions of the integrated pipeline.

Analysis Reporting

The process ID may require a status check for processing during the core executor stage (see above). This ensures that the workflow prior to analysis reporting has been completed successfully.

Database Check

As with other stages, mentioned above, the process_stages table can be queried for information on failed processing during analysis reporting:

```
SELECT * FROM process_stages  WHERE stage_name='analysis_reporter' AND stage_start IS NOT NULL AND stage_error IS NOT NULL;
```

The individual process IDs can be queried instead:

```
SELECT * FROM process_stages WHERE stage_name='analysis_reporter' AND process_id='XXXXXXXX';
```

Log File Check

Check the failed process IDs within the analysis_reporter.log file, in particular curl commands for uploading the analysis. Note any errors which have presented within the log file.

Directory Check

Check that the analysis and submission XMLs have been generated accordingly. These two files are required in order to submit analysis to ENA, not to mention the actual analysis files (see Core Execution - Processing Directory Check above).

Process Archival

Once identifying process IDs that have failed process archival, check the status of these during the previous stages.

Database Check

Information on process IDs which have failed the process archival stage can be obtained from the query below - similar to those above:

```
SELECT * FROM process_stages WHERE stage_name='process_archival' AND stage_start IS NOT NULL AND stage_error IS NOT NULL;
```

To query individual process IDs, use the following query of the process_stages table:

```
SELECT * FROM process_stages WHERE stage_name='process_archival' AND process_id='XXXXXXXXX';
```

Log File Check

A check of the process_archival.log file presents with errors to aid in troubleshooting. Search the process_id in question within the file to identify relevant error messages and processing.

Directory Check

In the case of process archival, check the process directory for the specific process ID. In addition to this, a successful process would result in the process directory being present within the archive directory.

Restart Processing

Restart Stage Processing for a Process ID

This section is relevant for restarting processes which have failed during data provision, core execution, analysis reporting or process archival.

Update the Database

To restart processing for a process ID at a given stage, alter the process_stages table only for a particular process_id and stage_name. SELECTA only progresses a process_id from one stage to the next, if the stage processing has been completed without error.

Example:

```
UPDATE process_stages SET stage_start=NULL, stage_end=NULL,
stage_error=NULL WHERE process_id='XXXXXXXXXX' AND stage_name='XXXXXX';
```

Restart Entire Workflow Processing

This section is relevant to restarting the selection to attribute stage or the entire workflow for a process ID.

Database Cleanup

For a given selection ID, a cleanup of the process_attributes table for a given process ID is may be beneficial to keep the table tidy, avoiding the addition of duplicate information.

Example delete of attributes for a selection ID to be restarted:

```
DELETE FROM process_attributes WHERE process_id IN (SELECT * FROM
process_attributes WHERE attribute_key='selection_id' AND
attribute_value=X);
```

Update the Database

To restart the entire workflow for a particular selection, alter the process_selection table. This enables for SELECTA to pick up selections and initiate the workflow by commencing selection_to_attribute. (See SELECTA Stage Components and Appendix 2 for more information).

Example:

```
UPDATE process_selection SET selection_to_attribute_start=NULL,
selection_attribute_end=NULL, selection_attribute_error=NULL WHERE
selection_id = X;
```

Example Troubleshoot Run Processing

Considering the following workflow:

Error presented within core_executor.log file → Error produced for process ID SRR2017653-120XXXXXXXX → Search for the LSF output and error logs for this process ID (found by the name of the log files) → LSF job did not complete successfully → Processing requires more memory → Check the corresponding process directory for the pipeline progress/error files → Restart SRR2017653-120XXXXXXXX processing accordingly

The above example investigates the stage-specific log file to identify the process ID. This process_id is used to search for the LSF output in the LSF-specific log file (named according to process_id). This presented with the error, however the processing directory - also named after the process_id - is checked for the progress/error files that are produced by the pipeline during pipeline processing. Finally the core executor stage for the process_id is restarted.

Common Errors

These generally fall into three distinct categories, those which are fastq-related and therefore can involve issues upstream from SELECTA processing. Secondly, system (or environment) related errors may occur, for example failure to generate the appropriate environment for SELECTA to run. The final common error that may be encountered includes pipeline processing errors. All require manual intervention, as they are variable in nature.

Appendix 1 - Properties File

The python scripts for each stage of the SELECTA workflow require an input properties file. This helps configure SELECTA and provides global variables which are utilised during the workflow. The properties file must include the following defined variables and parameters (note no variables should be in quotes):

Variable Name	**Variable Value**	**Example**
WORKDIR	Path to top working directory in which all processing is carried out. SELECTA creates additional directories within this processing directory, keeping data organised	
ARCHIVEDIR	Path to top archive directory in which all analysis for each process is kept in an archive folder	
DBUSER	Name of SELECTA database user	
DBPASSWORD	Password for the SELECTA database user	
DBHOST	Host for database	
DBNAME	Name of SELECTA database	
DBPORT	Database port number	
LSF	Define if cluster LSF commands can be used	To use LSF: LSF:YES/Yes/yes No LSF to use: LSF:NO/No/no OR LSF:
NPROC	Number of processors (CPUs) to available for SELECTA to make use of	
RMEM	LSF maximum memory to be provided per job, leave blank if no LSF	For 64GB memory: RMEM:rusage[mem=64000]

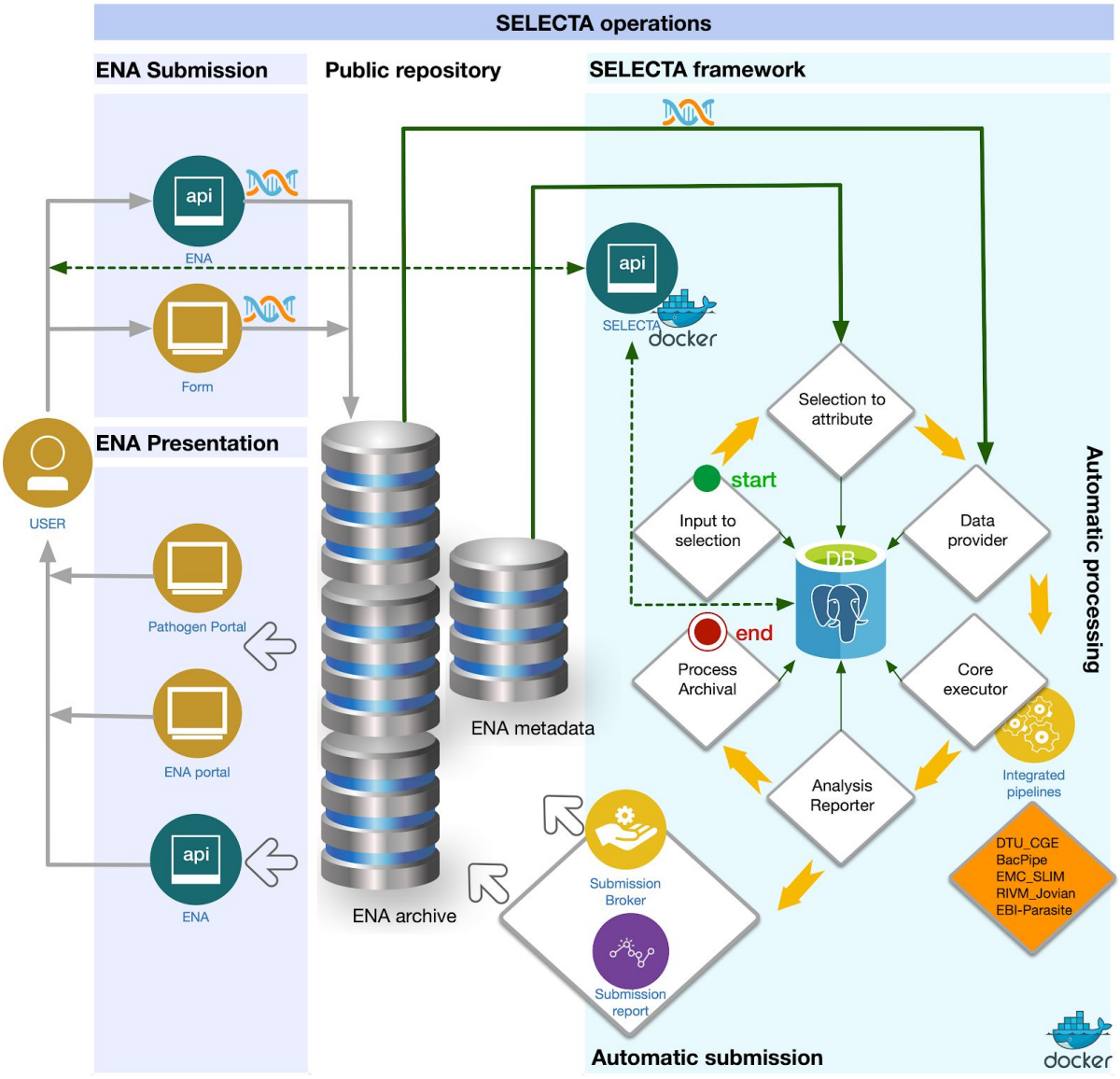
LMEM	Maximum memory to be provided per job	For 64GB memory: LMEM:64000
BGROUP	Group to submit interactive jobs to	BGROUP:/selecta
CGETOOLS*	Location of DTU_CGE pipeline container image used in running the pipeline	
SEQMACHINE	Sequencing instrument used	SEQMACHINE:Illumina
MAX_CORE_JOB	Maximum number of cores to be provided	
DTU_CGE_DATABASES*	Location of databases required to run the DTU_CGE pipeline	
DTU_CGE_VERSION*	Version number of DTU_CGE pipeline	
EMC_SLIM_PROGRAM*	Location of the program (python script) used to run EMC_SLIM pipeline	
EMC_SLIM_PROPERTY_FILE*	Property file required to run the EMC_SLIM pipeline	
EMC_SLIM_VERSION*	Version number of the EMC_SLIM pipeline	
SELECTA_VERSION	Version number of SELECTA system	
UAntwerp_BACPIPE_PROGRAM*	Location of python script running BacPipe	
UAntwerp_BACPIPE_VERSION*	Version number of SELECTA pipeline	
UAntwerp_BACPIPE_DEP*	Path to BacPipe folder	
PROKKA_PROGRAM*	Location of Prokka package	
RIVM_JOVIAN_BASE	Location of Jovian pipeline codebase	
RIVM_JOVIAN_VERSION*	Version of the Jovian pipeline being utilised	
RIVM_JOVIAN_PROFILE*	Path to Jovian configuration file	

ANALYSIS_SUBMISSION_MODE	Instance the analysis submissions are to be made	
ANALYSIS_SUBMISSION_ACTION	The action required for analysis submission	
ANALYSIS_SUBMISSION_URL_PROD	URL to submit analysis to ENA production	
ANALYSIS_SUBMISSION_URL_DEV	URL to submit analysis to ENA development	

* Required only if using SELECTA with COMPARE-specific integrated pipelines.

Appendix 2 - SELECTA Detailed Workflow

Overview



SELECTA Software Architecture - repeated for convenience

SELECTA algorithmic approach

The raw genomics sequences data submission follows the conventional data submission routes, shown above. Once submitted, the data are automatically identified and captured by the SELECTA framework for further processing. SELECTA relies on a rule-based approach, defined in its backend PostgreSQL database, to identify newly submitted raw data for processing. Data grouped by either datahub or project accession are annotated with bioinformatic tools that should operate on them. The system also monitors and controls, based on user defined parameters, whether processing in a group is an ongoing or a one-time operation. This implies that the same dataset can continually be processed by

multiple different analysis pipelines for example for the purpose of algorithm comparison on join-merge analysis. There are also circumstances where re-analysis is necessary. For example due to an error in sequencing or subsequent re-sequencing with new and better chemistry reagent. SELECTA rule-based selection makes it possible to define a rule that enforces re-analysis of the new sequence data by setting it's processing type and the continuity parameter, accordingly.

Operations

SELECTA processing goes through six sequential and dependent stages. The input to the selection, the selection to the attribute, the data provider, the core executor, the analysis submission and the data archival. All stages make use of the same configuration file. This configuration file defines the paths to the integrated analysis pipelines, its dependent databases, and parameters for load balancing if required. The Input to attribute stage is the first step in enacting the SELECTA workflow. It is done by amending new records into the SELECTA backend PostgreSQL database, such as data hub, study or run, data center, analysis pipeline name and other metadata such as continuity and process type. The table below illustrates SELECTA sequential and hierarchical modes of action.

Sequence Data Submissions (Datahub Reporting)

Sequence data is obtained from a path defined upstream in the metadata retrieved. This file path is used to obtain the read sequence file(s) from the ENA FTP repository. (Note read sequences processed can be single or paired end). The codebase can be altered to retrieve the sequence file(s) from an alternative storage system.

Read sequence files (runs) belong to specific projects, with these projects belonging to specific datahubs. Therefore a given datahub can have multiple projects with each of these projects consisting of many runs. A major advantage of SELECTA includes the ability to process these runs at each of those three levels - datahub, project and run, while maintaining the level of hierarchy. For example, processing runs from a specific datahub (e.g. dcc_allison) would involve obtaining all the runs from the various projects that have been submitted to dcc_allison.

To specify which level SELECTA should process sequence data at, alter the process_type (with the value 'datahub'/'study'/'run') for the specific selection within the process_selection table. Note if processing at the study (project), provide a valid study_accession value. If processing at the run level (which is housed in a datahub and project), provide a valid study_accession and run_accession.

Input to Selection

Input to Selection involves defining a 'selection' of data that require processing by a specific pipeline. This includes separate selection IDs for each datahub, datahub project or datahub project run to be processed by the specific pipeline, inserted as a new entry into the process_selection table. Note this is the sole manual stage in the SELECTA workflow (and

thereby does not include a set python script), in order to ensure appropriate SELECTA processing occurs. Additionally, this can be carried out through SELECTA's API and initiates the SELECTA workflow.

From Selections to Attributes

This section refers to the selection_to_attribute.py script and follows on from the previous steps.

The selection defined in 'Data Selection' translates to runs (belonging to projects within the datahubs) which have not been processed through SELECTA and require processing. Information from the process_selection table (characterised in class object called selection - refer to selectadb.py) is coupled with information within datahub metadata TSV files (retrieved from the ENA FTP repository) and stored within the process_attributes table. These 'attributes' for each process (run) are referenced through the corresponding table in a 'default_attributes' class object. Therefore the different attributes can be referenced/altered at different stages.

During this stage, an update of the process_stages table takes place to include new rows for each run (to be processed) and SELECTA stage. Therefore for each run to be processed, four new rows are inserted (one for each of the following: data_provider, core_executor, analysis_reporter and process_archival). This will be updated during the SELECTA workflow to track the processes. Additionally, the process_report table is also updated to include a start datetime stamp for the SELECTA processing of a given process_id (run which belongs to a project and datahub).

Data Provision

This stage refers to data_provider.py script and follows the previous stage. Sequence data must be downloaded in order to carry out analysis. Data provision carries out the download but also ensures sequence data is processed by the datahub-defined pipeline. This stage runs following a check on the process_stages table for runs (process_ids) which have not commenced the data provider stage.

From the list of processes yet to undergo data provision, its attributes are checked to find the location of the sequence files for each run (note this is a reference to the default_attributes class object). This is then used to download the sequence data from the ENA FTP repository, within the appropriate working directory (named after the process ID).

Core Execution

This stage refers to core_executor.py and follows data provision. The process_stages table is checked for runs (process_ids) which have completed data provision and are yet to undergo the core executor stage. Using this list of runs, SELECTA schedules the processing of the run through the datahub-defined pipeline, defined in attributes. For each integrated

pipeline, an associated class object exists. This aids in providing specific actions that may be required to facilitate its integration into SELECTA framework. These actions include building the command to execute processing of the fastq file(s) through the pipeline and post-processing of results, among others.

Pipelines are run through a built command within the pipeline class object. If LSF setting has been included within the properties file, then SELECTA makes use of this in processing. This stage is also parallelized using the Parallel python package to decrease processing times and moreover process as many runs as possible at once. If LSF is not included, then the pipeline processing is carried out as a shell command.

Following the pipeline processing, a post-process stage follows in order to prepare for the analysis reporter stage. This ensures that output formats remains uniform despite different pipelines being utilised in processing.

Analysis Reporting

This section refers to analysis_reporter.py. The process_stages table is checked for runs (process_ids) which have completed data provision and core execution and are yet to undergo the analysis reporting stage. Using this list of runs, analysis reporting is carried out on the runs, which involves submission of integrated-pipeline analysis to the ENA. Therefore SELECTA generates a submission XML to submit analysis which is defined in the analysis XML to report the analysis.

A submission class instance is produced which includes information required for and the actual building of the submission XML (refer to sra_objects.py). An analysis class instance is generated including information required for and building of the analysis XML.

Examples of methods for analysis presentation include through the ENA browser, COMPARE platform or Pathogens Portal. The requirement for desired presentation have been stated in Integrating Pipelines: Required Pipeline Input and Output.

Once the analysis has been reported appropriately, the process_report table is updated to include submission IDs for ENA submission of the analysis for a particular process ID.

Process Archival

To conclude the SELECTA workflow, an archive of the integrated-pipeline processing directories are stored. This section refers to process_archival.py. The process_stages table is checked for process_ids which have completed data provision, core execution and analysis reporting and are yet to undergo the process archival stage. Once the directory is archived, the process_report table is updated to include a datestamp of the completion of the workflow for a run.