Teemu Kataja

Designing and developing a data processing pipeline for archiving sensitive human data

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technologies

Thesis

9 March 2018

Metropolia

Author	Teemu Kataja
Title	Designing and developing a data processing pipeline for archiving sensitive human data
Number of Pages	30 pages + 20 appendices
Date	9 March 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technologies
Professional Major	Health Technology
Instructors	Ilkka Lappalainen, Research Data Services Manager, CSC Juha Törnroos, Software Architect, CSC & CRG Mikael Soini, Principal Lecturer, Metropolia

This thesis was done for CSC – IT Center for Science Ltd, which is Finland's national IT center and one of the 21 ELIXIR Europe partners. ELIXIR is an intergovernmental project of the European Union (EU) that aims to standardize practices and facilitate the creation of a modern and service-oriented bioinformatics infrastructure to enable scientists to do research more efficiently. The work done in this thesis is part of an ELIXIR project work package to create data submission tools for sensitive human data.

The aim of this thesis was to create a scalable and modular python workflow for the purpose of processing and archiving transmitted genomic datasets. The project produced a set of self-sufficient scripts that are used as modules in automated workflows that utilize a python library called luigi, which creates a convenient pipeline for a tasked dataflow. Before getting into the technical part of the thesis, the background of genomic research in Europe is introduced along with cumbersome practices that are currently in place.

The work consisted of designing and developing completely new software components that will be used in supercomputers to process and archive genomic datasets. Software design and development was conducted using CSC's cloud computing environment and version control was done using github. Due to the nature of EU projects the github repository that contains the produced scripts is public, and available to be viewed by anyone. Direct links can be found in the references and appendices.

The scripts created in this thesis were tested at the end of the project in mock-up end-to-end testing as well as in a real dataset transmission situation. The scripts were deemed to be well-functioning, and thanks to their well-documented nature, maintenance of the data processing pipeline will be easy. Future changes are also possible due to the nature of the modular object-oriented (OOP) design. This set of scripts will replace an outdated set of shell scripts that were used to archive datasets. The old scripts were not documented and were made hard to read and required the work effort of three maintenance engineers. Thus, this thesis creates immediate real value in reassigning work force, and the scripts are also available to all ELIXIR partners in Europe.

Keywords	Data processing, data archiving, tasked workflow, automated pipeline, genomic dataset, ELIXIR

Metropolia

Contents

List of Abbreviations

API	Application Programming Interface
AES	Advanced Encryption Standard
CSS	Cascading Style Sheets
CRG	Center for Genomic Regulation
CPU	Central Processing Unit
CVCS	Centralized Version Control System
DAC	Data Access Committee
EGA	European Genome-phenome Archive
EBI	European Bioinformatics Institute
EMBL	European Molecular Biology Laboratory
FTP	File Transfer Protocol
HDD	Hard Disk Drive
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
NeIC	Nordic e-Infrastructure Collaboration
OOP	Object-Oriented Programming

RAD	Rapid Application Development
RDBMS	Relational Database Management System
RES	Re-Encryption Server
REST	Representational State Transfer
VCS	Version Control System
VPN	Virtual Private Network

1 Introduction

Europe, as individual nations, has been lagging behind in genomics research compared to other developed countries [1, p. 7]. The extent of research and development can largely be associated with available capital. By comparing the amount of public funding spent on genome studies in different countries, European nations hold 5 places in the top 10, the top 5 list being as follows; United States (1 billion USD), United Kingdom (350 million USD), Canada (125 million USD), Japan (115 million USD) and China (80 million USD) in 2006. However, combining all the European nations, the amount of funding is close to that of the United States' at 937 million USD. [2, Table 4.]

The key to improving European genomics research is collaboration between member states of the European Union (EU), and other close non-member states. ELIXIR is an EU funded intergovernmental project that aims to connect 21 nations together in life sciences by standardizing the infrastructure and methodology of common practices [3]. ELIXIR consists of several work packages which will be later defined in Chapter 2 along with other notable project collaborators and the current situation the scientists in genomics are currently riddled with. By standardizing and facilitating a common bioinformatics infrastructure over Europe, ELIXIR and the contents of this thesis aims to enable life science researchers to initiate studies more easily, efficiently and at a faster pace. The holistic vision is to produce more opportunities for innovation and discoveries that lead to the improvement of human life.

The aim of this thesis was to create a well-documented and scalable data processing pipeline that can be automated and distributed over several supercomputers. The pipeline is a set of scripts working together to create a workflow that is capable of handling massive amounts of data in the magnitude of several terabytes per day. The created program processes received genomic datasets from research institutions and biobanks into a secure archive. The detailed function of each script and operation of the overall process is explained in Chapter 4. In addition to this, a web portal was devised, that could work as the interface for ELIXIR Finland Node. The web portal was equipped with a user portal for scientists to request (and in the future to submit) genomic datasets, as well as an administrator portal for the maintainers of the genomic database. This thesis however, is limited only to the development of the data processing pipeline.

This thesis was done for CSC – IT Center for Science, which is a non-profit specialist organization funded by the Finnish government and owned by the state and Finnish higher education institutions. CSC's mission is to provide IT infrastructure, tools, education and support for academic institutions and scientific research in Finland. [4.] CSC is Finland's ELIXIR Node and contractor for the EXCELERATE (life sciences project) work packages, and as such, was tasked with creating a scalable data processing pipeline to be used as a tool for archiving sensitive human data. The work done in this thesis will replace an outdated and cumbersome piece of software, thus relieving three maintenance engineers to be used in other tasks due the program's simple, automated and well-documented nature. The modularity of the program workflow also allows it to be easily modified and distributed to ELIXIR partners.

2 Background

2.1 ELIXIR Project

As briefly mentioned in Chapter 1 Introduction, ELIXIR is an EU funded intergovernmental life science project that aims to connect the genetic scientists of Europe together in order to boost the efficiency and rate of research. EXCELERATE is ELIXIR's current project for the years 2014-2020, and it's funded by the 80 billion euro Horizon 2020 research and innovation programme. Of the large Horizon 2020 budget, 19,1 million euros are accredited to the development of the ELIXIR infrastructure. [5, p. 5; 6, p. 2-7].

The ELIXIR network currently consists of 21 member nations, that are marked in Figure 1. The orange cylinders on the United Kingdom (UK) and Spain represent the European Genome-phenome Archives (EGA), that are the central hubs for European biological data. They are hosted by the European Bioinformatics Institute (EMBL-EBI) in UK and Center for Genomic Regulation (CRG) in Spain. The other 19 blue cylinders represent the ELIXIR Nodes that host the smaller, national biobanks (Local EGAs). In the past European research institutes submitted their research data directly to the EGA, but in the future scheme national ELIXIR nodes will serve as smaller, local biobanks, that intake research data from within their respective borders. When the ELIXIR infrastructure is completed, researchers will be able to browse and request genomic datasets from any ELIXIR node, and thus scientists have access to a much broader pool of research data.

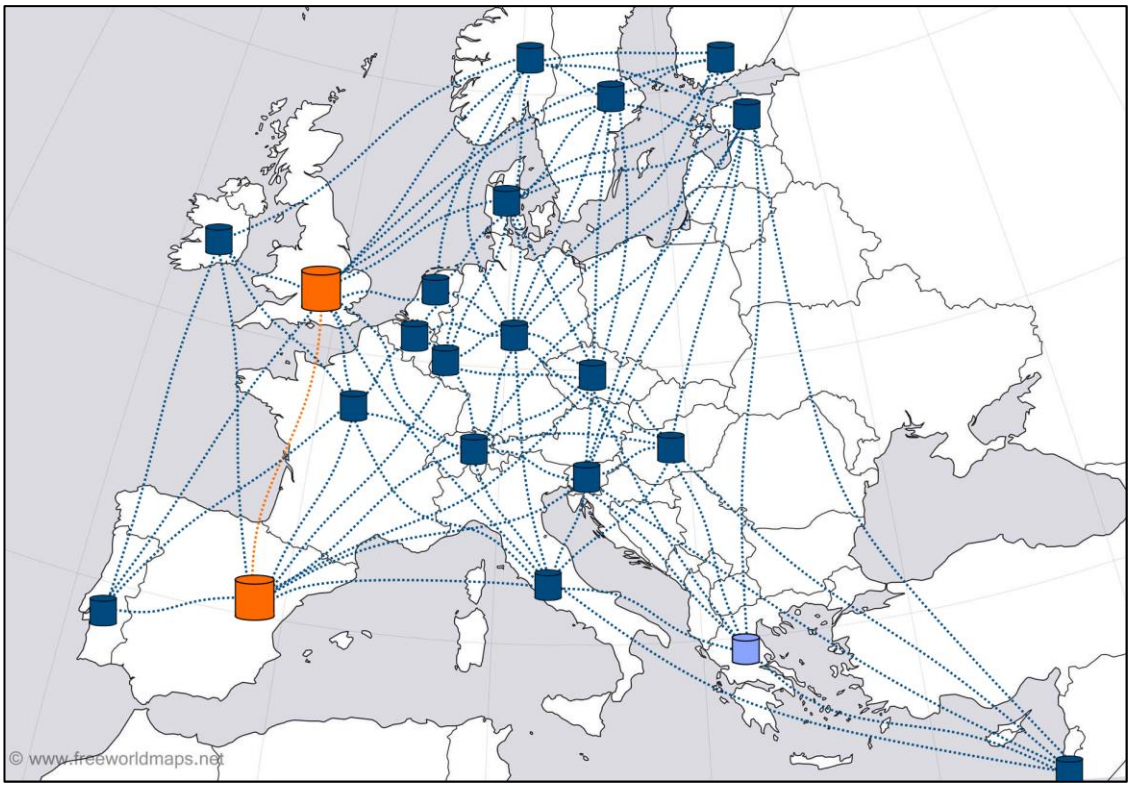


Figure 1. ELIXIR Nodes marked on the map of Europe [7]. Illustration. EGAs marked in orange, national ELIXIR nodes marked in dark blue and Greece's ELIXIR node marked in light blue (observer, not full member).

2.2 ELIXIR Project Collaborators

Because the *to-be* life sciences infrastructure in Europe is such a large task to be done, the work is divided into several work packages, of which work package 9 encompasses the design and development of this thesis, *"Task 9.3: EGA in the Clouds, Large scale data mirroring and syncing"*. During the design and development of this data processing module, regular video conferences were held at a weekly basis with ELIXIR partners from Finland, Sweden, Spain and the UK. The aim of the weekly meetings were to gather feedback from collaborators in order to design and develop coherent software and to maintain a high level of conformity. The many work packages in EXCELERATE are tightly knit together, so workers from different projects have to communicate with each other regularly. The development of this module was closely related to work package 4, which covers the secure authentication of users into the genomic database. [8, p. 4-5.]

2.2.1 EMBL-EBI

The European Bioinformatics Institute (EBI) is part of the European Molecular Biology Laboratory (EMBL). It's an international organization whose purpose is to promote innovation and research in the field of biology. It can be thought of as the precursor to ELIXIR, as the central biobank in Europe (EGA) is hosted at EMBL-EBI. Even though EMBL-EBI is located in the UK and serves as one ELIXIR node in the European life sciences network, the United Kingdom has its own national ELIXIR node, making UK the only European nation with two ELIXIR nodes in one country. As such, EMBL-EBI also engages in the development of EXCELERATE projects. [9.]

For ELIXIR EXCELERATE, the partners from the British ELIXIR Node have developed certain encryption and decryption tools using HTTP-requests (Hypertext Transfer Protocol), that were used in the development of this data processing pipeline module. The tools developed by the British ELIXIR partners work by sending data to an online service via internet, which will return a processed result. The mentioned online service will be further introduced in Chapter 4.4.2 Res.py, as it's an essential component in the data processing pipeline that was developed in this thesis.

2.2.2 CRG

The Center for Genomic Regulation (CRG) is part of Spain's National Bioinformatics Institute, and as such, closely resembles the EMBL-EBI. CRG acts as one of six ELIXIR collaborators in Spain, and therefore is listed here as one participant in the development of this module, as they have taken part in regular project meetings where the development of this thesis was discussed. The main technical instructor to this thesis who works at CSC moved to Spain to continue working on ELIXIR EXCELERATE from CRG's sector. It should also be mentioned, that CRG hosts the second central biobank in Europe (EGA) in addition to EMBL-EBI. [10; 11.]

2.2.3 NeIC

The Nordic e-Infrastructure Collaboration (NeIC) is an initiative established in 2012 to enhance the research and development of IT infrastructure in the Nordic countries. The organization is formed by each Nordic country's national IT center (for Finland that would be CSC). The aim of the organization is to discuss common practices and to facilitate the development and operation of high performance supercomputers in the Nordic area. The Nordic co-operation allows the sharing of data storage and computational capacity over the Nordic university networks. [12.]

2.3 Current State and Future Vision of Initiating Research

A genome is the complete genetic information of a human (full DNA), containing all the genes that make up that individual. It's comprised of over 3 billion nucleotide pairs of Adenine, Thymine, Cytosine and Guanine that encode the functions of the human body with roughly 20 000 to 25 000 genes. The first full sequencing of this long chain was completed in 2003 by the Human Genome Project after 12 years of research, and the so-called reference human genome is currently available to be browsed by anyone at EMBL-EBI's database (refer to *Reference number 14* for browsing the human genome). [13, p. 3-4; 14.]

2.3.1 Current State

The work done by life science researchers is currently hindered by bureaucracy and inconvenience. Figure 2 illustrates the many steps a scientist must take in order to begin studying the genetic makeup of humans. If a life science researcher wishes to conduct studies on completely new biological data, they first must request the physical (biological) samples from a biobank and have the owner of the sample to approve the usage of the sample for research purposes. Human genetic information is considered sensitive data, and therefore is heavily protected by security laws and regulations, because individuals could be identified from the biological samples. Typically, any identifiable data is redacted from the genome before publication. [15.]

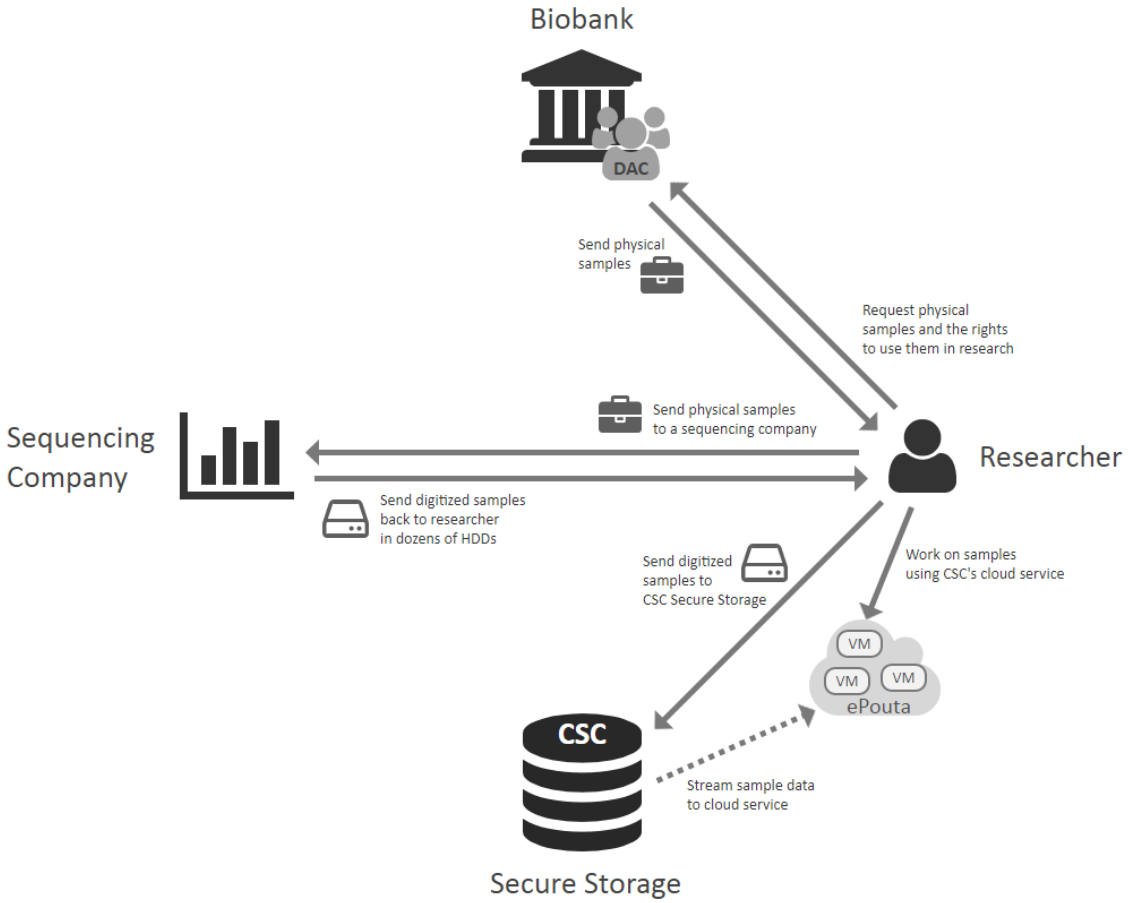


Figure 2. Current situation of initiating research on new data.

If the scientist is granted access to the research use of the biological samples, they will be shipped to the researcher for sequencing. Genome sequencing requires expensive equipment, and the samples are often sent offshore to be handled by a third party company. After sequencing the biological samples, the genetic information is in digital form. Because the human genome is so immensely large and samples for genomics research are often gathered from hundreds if not thousands of subjects, the resulting amount of data is problematic and the methods of data storage cause problems for scientists. [15.]

The digitized samples are sent back to the researcher in hard disk drives (HDD). It's not uncommon for the scientist to receive 50 HDDs due to a large amount of research material. When this happens, the researcher is unable to utilize the genomic datasets, because it's impractical to plug in 50 HDDs to a single computer. The HDDs are then sent to CSC to be stored in the Scientific Cloud Computing Environment. By utilizing the cloud services of CSC, scientists are able to exploit high-performance computers (supercomputers) in their research and calculations. Genome studies are a painstakingly long and

time-consuming process, due to the nature of large data masses and the need for hefty computational functions. [15.]

2.3.2 Future Vision

The future vision of enabling genomics research is a fully optimized and service-oriented network of ELIXIR partners, as presented in Figure 3. The aim of the network is to make it easier for scientists to commence life science research, thus generating new analyses and investigations into the human genome and to add value into gathered biological samples.

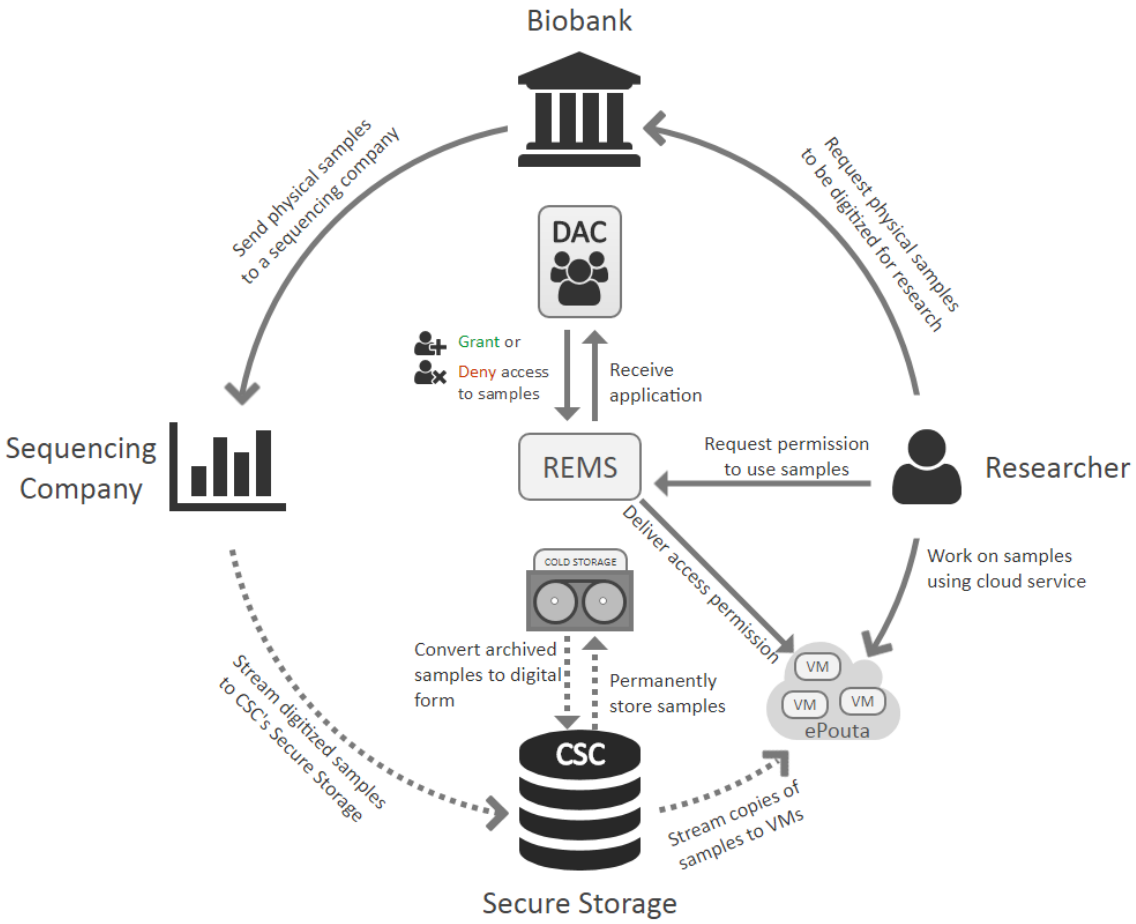


Figure 3. Future, *service-oriented*, vision to ease the work of life science researchers.

The service-oriented approach would eliminate cumbersome tasks from the researcher, enabling them to focus more on the task at hand. The new system would support the fluid usage of old and new genomic information. If the desired genomic dataset is already available at any ELIXIR node, the scientist could simply request access to use it from a

Data Access Committee (DAC) who owns the dataset. The application would be sent through an online portal (Resource Entitlement Management System, REMS) directly to the owners for review. Another added benefit to the new identification system (REMS) is a vision of granting *bona fide* statuses to trustworthy scientists that have been peer-reviewed. Such a status would allow a researcher to initiate research on a dataset without the need for requesting permission, thus enabling faster studies. [15.]

After given permission to use a new biological sample in genomics research, the DAC or the scientist would relay the information to the biobank or institution that is holding the sample. The biobank or institution would then send the sample directly to a partner sequencing company, that would stream the digitized genomic data straight to CSC's secure servers. [15.]

When the digitized sample is received on CSC's servers, it's ready to be studied via CSC's Scientific Cloud Computing Environment, to which the researcher's permissions are delivered from REMS. Such a service-oriented approach to enabling life sciences research would make it easier for scientists to start new studies and to utilize the cloud services and high-performance computers at CSC. There are roughly half a million life science researchers in Europe, and all of their work can be connected to each other via the ELIXIR network. The rate and efficiency of research is expected to increase dramatically after the infrastructure has been completed and adopted by the experts. [15.]

3 Developer Tools

3.1 Git Version Control

A version control is any system that holds a record of modifications to a file and can be used to create and load saved back-ups from past dates. They are especially useful in software development, where it can be necessary to return to an older stable version of the code in case fatal errors occur in the program. Version control programs come in different operative principles, and for this thesis a Centralized Version Control System (CVCS) called Git was used. In a centralized type of version control system several developers can contribute to a single project, because the source files are stored in a common virtual database. [16, p. 5-7.]

Git was especially useful in this project due to the need to peer-review and distribute the produced scripts for ELIXIR partners. The created program developed in this thesis is freely available to be downloaded by anyone via GitHub. The public repository can be downloaded using git by using example code 1. [16, p. 21; 17.]

```
git clone https://github.com/CSCfi/lega-mirroring
```

Example code 1. Git command to download the repository of the data processing pipeline source files which were developed in this thesis.

3.2 Python Programming Language

Python is a high-level programming language, meaning that it is easy to read and write by a human, but must be translated to machine language before it can be interpreted by a computer. It was selected for the development of these data processing modules by the ELIXIR partners due to its simplicity and modularity. Python can be enriched with third party libraries and it supports object-oriented programming, which was heavily utilized in the automated workflow process introduced in Chapter 4.5 Development Phase: Automated Pipeline Workflow. [18; 19.]

Python was a great choice for the programming language of this work package, as it can be exploited by Rapid Application Development (RAD) methodology. Because Python scripts are not compiled (e.g. compared to Java and C), modifications to code can be tested extremely fast, and debugging is effortless. [18.] For example the first primitive, but working, version of *monitor.py* (introduced in Chapter 4.4.1) was created in mere 4 hours after receiving the requirement specifications for the modules.

3.3 MySQL Database

MySQL is a type of Relational Database Management System (RDBMS), which is a software used to store data of different types [20, p. 1]. In this project it was used on two occasions: by the data processing pipeline scripts to track the status and location of files as well as by the automated workflow to fetch identity and verification data.

In the beginning of the development, a local MySQL server and Python-client was used to develop and test the scripts. This approach was chosen for convenience, as it required no external database connections over the internet. In time, when the project matured, a MySQL server was set up in the cloud environment, so that testing over other computers could be conducted. In the final version of the finished product a PostgreSQL database server is used. The similarity of syntax between the two are so close, that no modifications were needed during the transition from system to system.

3.4 AXURE RP8 Prototyping Tool

AXURE RP8 is a commercial prototyping tool (however, a free student license was used in the design phase of this thesis). It can be used to create dynamic and interactive flowcharts, wireframes and clickable prototypes. Its consistent framework makes it easy to learn, and it can be used to quickly visualize and bring ideas to life. [21.]

AXURE was used to polish and re-draw concepts from paper to electronic format in order to more efficiently spread information to project partners. Most of the figures presented in this thesis were drawn using AXURE. It was also utilized in the design of the web portal, which is left out of this thesis. However, it should be mentioned, that AXURE was a useful tool in creating a clickable website prototype.

3.5 Shell Scripts

Shell scripts are commands that are directly inputted to the command line on a Linux text-based interface [22]. The CSC Scientific Cloud Computing Environment is used via a Virtual Private Network (VPN) connection and is controlled using a command line window on the working computer. Learning shell scripts was an important task in the development of this thesis, as they were constantly used in the testing and real operation of the data processing modules. Shell scripts to execute each module and workflow are further introduced in each script subchapter under Chapters 4.4 and 4.5.

3.6 CSC ePouta Scientific Cloud Computing Environment

ePouta is a cloud computing environment hosted by CSC designed to be used when working with sensitive data that requires secure servers and storage. It's an Infrastructure as a Service (IaaS) platform, that is hosted on the national supercomputers. Scientists can apply for a slice of the computational capacity of the high-performance computing facilities CSC hosts, and set up a tailormade cloud environment. [23.] The development of this thesis was conducted in the mentioned cloud environment.

4 Data Processing Pipeline

4.1 Introduction to Software Operation

This chapter introduces the design and development of the created software modules (data processing pipeline) and explains the operation of each script (module) in detail. The process overview is presented in Figure 4, and briefly introduced in the next paragraphs. Thorough explanations for the python scripts are then given in chapters 4.4 and 4.5.

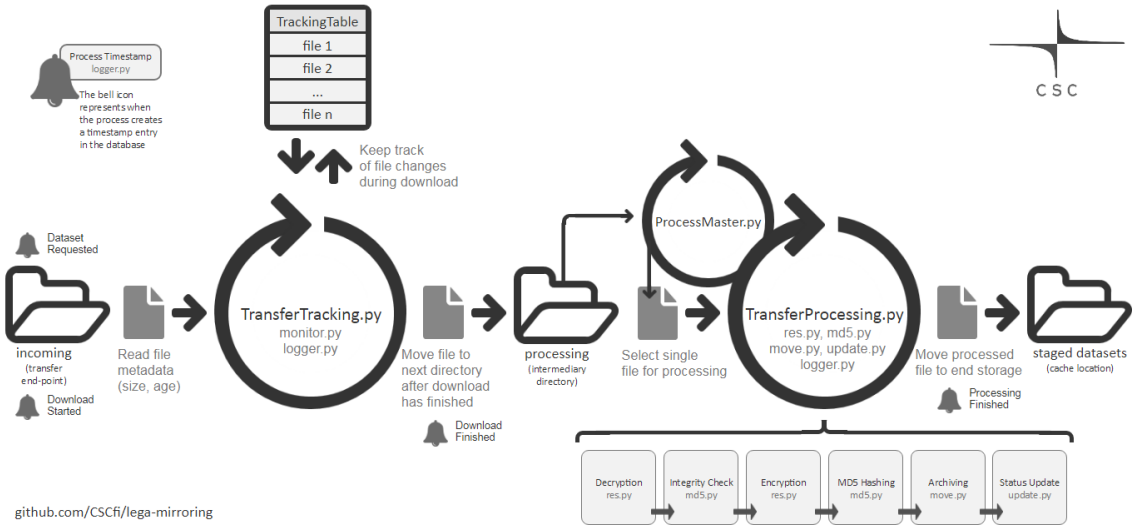


Figure 4. Data processing pipeline overview picture presenting the operation and steps taken for processing and archiving sensitive human data. Enlarged version can be seen on Appendix 1.

When a genomic dataset is requested, timestamp from the event is marked on a tracking table. The second timestamp is added when download of the datafiles begin. The timestamps are managed by *datasetlogger.py*. The process starts from the left-hand side of Figure 4, in the directory titled *incoming (transfer end-point)*. Files in this directory are investigated by an automated workflow script called *TransferTracking.py*. *TransferTracking.py* is an automated workflow that consists of two component scripts: *monitor.py* and *datasetlogger.py*. Monitor.py is a script, that determines if a file has finished downloading by keeping track of file size and age. When *monitor.py* determines that a file has finished downloading, *dataselogger.py* will timestamp the event and the file will be moved to the next directory titled *processing (intermediary directory)*.

The actual data processing and archiving begins in this directory. The processing directory is monitored by a second automated workflow called *ProcessMaster.py*. *ProcessMaster.py* generates list of contents of the processing directory accordingly to given parallelization parameters and selects files for processing. When a file has been selected by the master workflow, it will produce child processes called *TransferProcessing.py* for each selected datafile. This final automated workflow consists of several component scripts: *res.py, md5.py, move.py, update.py* and *datasetlogger.py*.

The main processing workflow, *TransferProcessing.py,* begins its operation by decrypting the datafile using *res.py.* The integrity of the decrypted file contents are then verified by the script *md5.py*. If the datafile transmission was successful and the file is intact, it will be re-encrypted using *res.py* again. A new checksum hash is then generated for the newly re-encrypted datafile using *md5.py*. After completing these computationally hefty processing steps, *TransferProcessing.py* will use *move.py* to archive the re-encrypted datafile along with its newly generated checksum hash to a secure storage. Finally *update.py* will update the status and archive location to the tracking table for easy access.

4.2 Design Phase

This chapter explains and presents the given requirement specifications and the methods of design. The project initiated by receiving the project overview and software requirement specifications, following these instructions the design process was started and the practical part of coding (development) soon began as well. This thesis was conducted

using agile methodologies of rapid *design → development → testing → return to design* style of working.

4.2.1 Project Requirements

At the beginning of the project, a meeting was held to discuss the software requirement specifications, and the data transmission structure. In a given example situation a genomic dataset is sent from EMBL-EBI to CSC via GridFTP. The duration for transmitting a 100 TB file set is expected to take 2 weeks. When the files are received at CSC's end, the data processing pipeline would start automatically and begin to process and archive the received files. [15.]

The requirement specifications for the software were listed as follows (Appendix 2):

1. Check if datafile transmission is completed
2. Check integrity of transmitted files
3. Decrypt files
4. Check the integrity of decrypted files
5. Re-encrypt files using CSC's encryption key
6. Compute new verification checksum (md5) for files
7. Archive files to designated location
8. Update file status and archive location to tracking database

Reading from the list above; the created software should be able to know when a file transmission has been completed, so that it doesn't process and archive incomplete files. It should also be able to check the integrity of the files to verify, that the received file has remained intact. The software should the be able to decrypt the file and then verify the integrity of the contents. After this, the software should be able to re-encrypt the file and compute a new checksum hash for it. Finally the software should be able to archive the files to a predefined location and update their status and new location to a tracking table in a related database. The benefits of agile software development (rapid testing) soon

showed, that step 2 could be removed from the initial requirement specifications, and the finished product would have fewer bottlenecks (heavy computational steps).

4.2.2 Design Methods

Due to the nature of the project, compositional design strategy was used. The principle of compositional design strategy is to identify the pieces or modules that make up the whole, and to start working by building the software up from small pieces. When the individual modules and pieces are finished, they are put together to form the complete product. This method of dividing work into tasks made the design and development process manageable and easily executable. [24, p. 61-62.]

The creative process for designing new modules (starting work on new tasks) began by drawing the operation, important functions and properties on paper. An example draft for the creation of one script (monitor.py) is seen in Figure 5 below. Each script was designed by drafting detailed instructions and workflows, so that when starting the development stage of the design process, it was easy to look at the draft and immediately know what to do next.

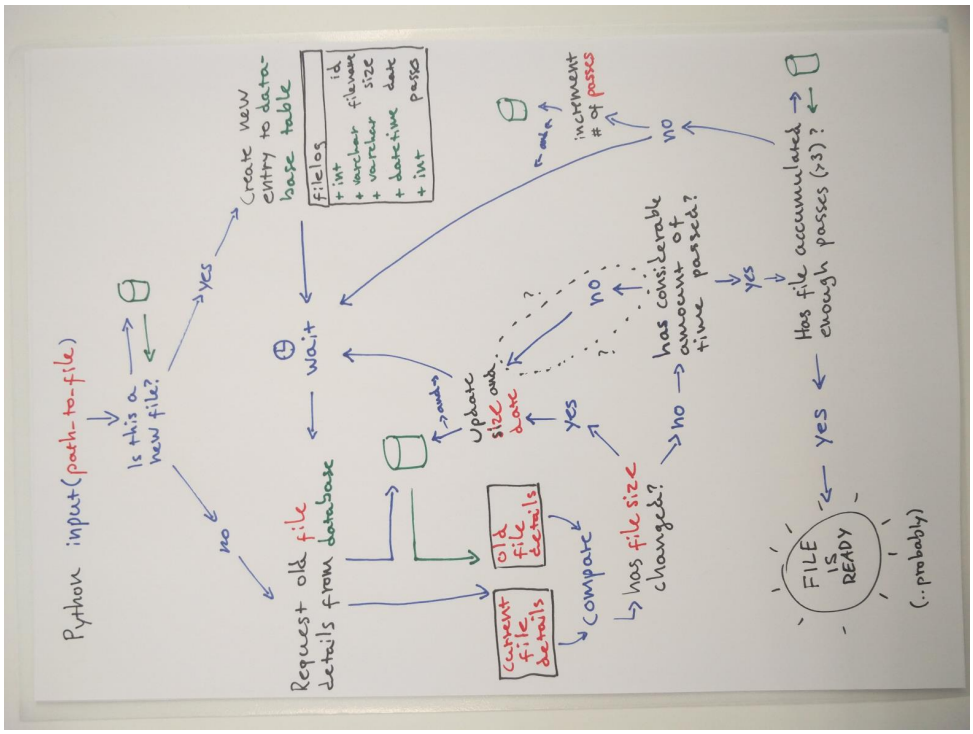


Figure 5. Example draft depicting the operation of monitor.py. An enlarged version can be seen on Appendix 3.

4.3 Development Phase: Creating a Database

A MySQL database server was set up, which served as a tracking tool to follow the file processing and archiving operations. The database structure was a relational database with interconnected tables containing specified file and time properties. The database structure is presented visually in Figure 6, and the database creation script can be viewed on Appendix 4. MySQL was used in the development phase, because it's easy to set up and use, but the production machine uses a PostgreSQL database. Fortunately the syntax of the two database systems are almost identical, so no modifications were needed after implementation of the database from test to production machine.

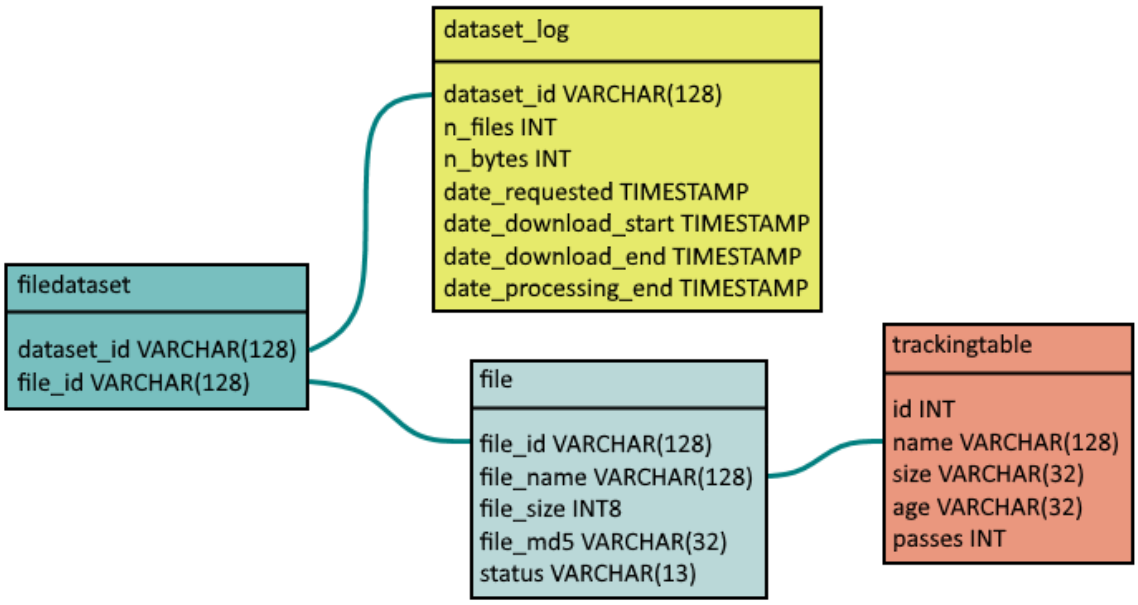


Figure 6. Database structure for tracking transmission and archiving of genomic datasets.

The table titled *filedataset* is the common bounding table, which contains the highest level identificators; *dataset_id* and *file_id*. At this point it should be mentioned, that because genomic datasets are enormous (from gigabytes to terabytes), they are made up from several datafiles. Each dataset has a unique international identity number and associated datafiles that relate to that dataset id are also uniquely named.

The *file_id* in table *file* refers to *file_id* in table *filedataset*. This means that entries in table *file* present datafile properties and entries in *filedataset* present which datafiles belong to which dataset. Furthermore, *file_name* in table *file* refers to *name* in table *trackingtable*, which is used in the monitoring progress of transmitted files. The variables

file_name and *name* refer to the complete path (location) of the file, e.g. C:\User\directory\file.bam. The yellow table, *dataset_log*, is used to create timestamps of different events from the pipeline, and can be used to find missing datafiles and issues that have emerged. All four tables are managed by the script *datasetlogger.py*, which will be introduced in Chapter 4.4.7.

4.4 Development Phase: Self-Sufficient Scripts

This chapter introduces the component scripts (modules) that make up the workflow pipeline. The self-sufficient scripts were created to be used autonomously via shell commands, and to be used as components of more sophisticated automated workflows. The autonomous aspect gives the scripts more utility, as they can be used in problem situations to execute a specific task or step of the workflow. This modularity principle adds value to the created software, as it also makes the whole process more easily repairable.

The scripts must be configured before they are usable. An external configuration file *config.ini* must be edited with the correct variables, such as working directories, database credentials, API credentials and other important properties before any scripts can be utilized. The configuration file will not be discussed any further, as it contains several configurable variables, and it's already equipped with guideline comments that aid the user to fill in the form. The configuration file can be seen on Appendix 5.

4.4.1 Monitor.py

Monitor.py is the transmission checking script, that is run by TransferTracking.py workflow (Figure 4). A visual representation for this script can be seen in Figure 7 below, which is based on the draft on Appendix 3, and the complete source code for the script file can be viewed in Appendix 6.

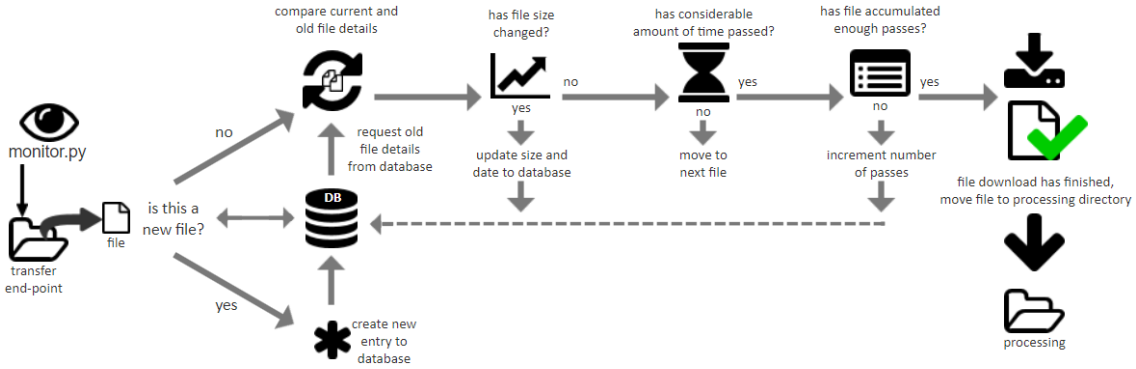


Figure 7. Flowchart diagram of the operation of monitor.py, based on the draft in Appendix 3.

When the script is run, it generates a list of the contents (files and subdirectories) inside the transfer end-point directory, which is read from the configuration file (Appendix 5). The process then begins by establishing a database connection for keeping track of file changes. When a file from the list is taken into examination, monitor.py will query the database to see if the file in question already has an entry in the database. If no matching file names are found from the database, monitor.py knows that this is a new file, and will create a new entry in the database. Upon encountering new files and adding them to the tracking table, the process ends for that file, and monitor.py will move on to the next file in the list.

As monitor.py is run again and again, it will eventually happen that it encounters a file that has a matching name in the tracking table. When this happens, monitor.py determines that it has encountered this file before, so its not a new file. In this case the process will proceed investigate changes in the file metadata. Monitor.py will read the file size in bytes and date of last modification from the file and request the same data from the database tracking table. Upon receiving the information from both sources, the current file and the details from the database, monitor.py compares the values. If the file size has changed, monitor.py will end the checking process at that point, because a change in file size indicates that the file is still being transmitted and is not ready to be processed further. If the file size hasn't changed however, monitor.py will look for the date of last modification. If the age of the file has remained the same for a defined period of time, e.g. 5 minutes, monitor.py will mark a passed run on the tracking table for that file. Upon accumulating a defined amount of passes, monitor.py will move the file from the receiving directory to the processing directory.

From the operating principle described above, it's apparent that the script must be run many times over a long time period to assure the completion of file transmission. Monitor.py is part of the automated workflow procedure, but can also be run autonomously as a single script, due to the built-in command line feature. The file transmission check is run with a simple command presented on Example code 2.

```
monitor /path/config.ini
```

Example code 2. Shell command to initiate a single autonomous run of monitor.py in predefined directory configured in configuration file config.ini.

4.4.2 Res.py

Res.py is a multifunctional decryption and encryption script that determines the course of action accordingly to given parameters. It utilizes ELIXIR's cryptographical microservice, the Re-Encryption Server (RES), which is run in the same cloud as the data processing pipeline. RES is another product of ELIXIR EXCELERATE and forms a part of this thesis. RES uses a 256-bit AES encryption protocol (Advanced Encryption Standard) that allows the creation of strong encryption. The full source code for res.py is presented on Appendix 7. [25.]

The operation of res.py is located in the TransferProcessing.py workflow structure as presented in Figure 4. It's used in the first and third tasks of the third luigi workflow, Decryption and Encryption. Res.py functions by making a REST API call to the active RES microservice and streaming the file contents to the server, which must be processed. This distributed microservice structure spreads computational capacity over several computers, so that not all processes are taken on the storage device. Like monitor.py, res.py is a part of the whole workflow, but can be run as a single script with a built-in command presented on Example code 3 below.

```
res <method> <path/file> <path/config.ini>

res decrypt file.bam.gpg config.ini

res encrypt file.bam config.ini
```

Example code 3. Shell command to initiate a single autonomous run of res.py with given method of decryption or encryption, targeted to a specified file with the path to configuration file.

The res command takes three parameters; operation, datafile location and configuration file location. When res.py decrypts an encrypted file, e.g. file.bam.gpg, it streams the encrypted data to RES microservice, which then streams the decrypted data back to the machine to a file named file.bam. And vice versa, when res.py encrypts a file, e.g. file.bam, it streams the un-encrypted data to RES microservice, which then streams the re-encrypted data to a file named file.bam.cip.csc. The file extension is not the same for differentiation purposes, and the .csc tag is added to clarify that CSC's encryption key was used.

4.4.3 Md5.py

Md5.py is another multifunctional script created for this program. It uses a message-digest (md5) algorithm to verify the integrity of transferred files and to create checksums for archived files. The md5 algorithm takes a string of any length and produces a 32 character long hash using a 128-bit encryption formula. Due to the nature of md5, it can be used to authenticate the contents of a string (that nothing has changed), because it always produces the same hash if given the same input. Therefore, md5 is useful in the verification of file integrity, i.e., to check if a single bit has changed in the file during transmission. [26.]

Md5.py is used in the TransferProcessing.py workflow (Figure 4) in tasks Integrity Check and Checksum Hashing. Md5.py can also be run as a single script using the built-in md5 command that takes three parameters similarly to res.py presented in the previous chapter. The single runs of md5.py are presented in Example code 4 below and the full source code is available on Appendix 8.

```
md5 <method> <path/file> <path/config.ini>

md5 check file.bam.gpg config.ini

md5 hash file.bam.cip.csc config.ini
```

Example code 4. Shell command to initiate a single autonomous run of md5.py with given method of checking or hashing, targeted to a specified file with the path to configuration file.

The md5 command has two possible methods, check and hash, that either verify the integrity of the transmitted file or creates a new checksum for the given file. When used to verify the integrity of a file, md5.py uses the check parameter and computes an md5 hash for the given file. Then the script queries the tracking database for the md5 hash that was received in the download manifest and compares the two. If the hashes match, the file was received correctly. To create a new checksum for a file, the hash command is used instead. This process will do the same hashing calculation for the file, but additionally print the 32 character long hash to a file bearing the name <input_file>.md5.

4.4.4 Move.py

Move.py is the archiving script in the data processing pipeline. It's the second to last step in the TransferProcessing.py workflow (Figure 4) just before update.py. The script is a relatively simple file mover, but it also prunes obsolete working files that have been left over from the data handling processes. If the script moves a file within the same file system, the move is atomic (happens practically immediately) as it only changes the path for the file.

The source code for move.py can be seen on Appendix 9, and a single-run command to use the script outside the workflow pipeline is presented on Example code 5 below. Move.py takes parameters for the datafile to archive, along with its associated .md5 file that contains the checksum, and the location of the configuration file.

```
move <path/datafile> <path/md5file> <path/config.ini>
```

Example code 5. Shell command to initiate a single autonomous run of move.py, given the parameters of files to move with the location of the configuration fi.

4.4.5 Update.py

Update.py is the last script used in the data processing pipeline, it can be seen at the last stage of TransferProcessing.py in Figure 4. Its role is to modify the tracking database to change the status and archive location of the processed files. When datafiles are downloading or processed, their status in the tracking database are seen as *pending*. Update.py changes that status to *available*, and then fixes the path name of the file from the receiving (transfer end-point) directory to the archive location (end-storage). The full

source code for update.py is presented on Appendix 10, and an example code to run the script outside the workflow is presented below on Example code 6.

```
update <file_basename_in_db> <path/config.ini>
```

Example code 6. Shell command to initiate a single autonomous run of update.py in order to manually update the status for a datafile.

A maintenance engineer might want to update the status of a datafile manually if problems have occurred in the pipeline that has caused the process to cease functioning. The update operation is run by giving the update.py script the basename of the file, which is the unique datafile id, and the location of the configuration file.

4.4.6 Dsin.py and getmetadata.py

Dsin.py and getmetadata.py are two utility scripts that are used to insert dataset metadata into the tracking database. Dsin.py is a script to read and insert metadata from text based formats, for when datasets are submitted to EGA from foreign sources, while getmetadata.py utilizes the EGA API to fetch dataset metadata from EGA directly. The two scripts are presented in Appendices 11 and 12 respectively.

Dsin.py operates by giving the script two parameters; metadata manifest and location of configuration file. Dsin.py then opens the text file and inputs the contents to the tracking table in the database. The manifest is formatted in the following manner: *filename*\t*md5*\n, which structures the information so that on each line there is filename on the left and the md5 checksum after a tab. An example shell command to run dsin.py is presented below on Example code 7.

```
python3 dsin.py manifest.txt config.ini
```

Example code 7. Shell command to read dataset metadata from an external text file and to insert it into the tracking database.

Getmetadata.py is similar to dsin.py, but instead of reading dataset metadata it connects to the EGA API via a REST call, and thus the metadata is relayed to the script via an internet. The received HTTP response is then inserted into the tracking database. The structure of this response is in JavaScript Object Notation (JSON) format, which is easily

computed with python. Because the information is requested from an online service, the dataset id must be given as a parameter for the script. The shell command to run getmetadata.py is presented on Example code 8 below.

```
python3 getmetadata.py EGAD00010014288 config.ini
```

Example code 8. Shell command to fetch dataset metadata from EGA API using an example dataset id.

4.4.7 Datasetlogger.py

Datasetlogger.py is a utility script that is used as part of other scripts in the workflow to timestamp important events to the tracking database. It's executed first at the time when a dataset is requested for downloading either by getmetadata.py or dsin.py in real transmission scenarios or by RequestDatasets.py (further introduced in Chapter 4.6 Testing and Evaluation) in testing phase. The full source code for datasetlogger.py can be seen on Appendix 13.

The timestamping events can be seen in Figure 4 (Appendix 1). The four tracked events are: *date_requested*, *date_download_start*, *date_download_end* and *date_processing_end*. The first date is timestamped upon requesting the dataset by either dsin.py or getmetadata.py, as said in the first paragraph. Then monitor.py handles the timestamps of the beginning and end of downloading. The timestamp for ending the download can also be thought of as the timestamp for initiating processing, as monitor.py moves the datafiles to the processing folder. The script is last used by update.py after processing has finished and the dataset details are updated to the tracking database.

4.5 Development Phase: Automated Pipeline Workflow

This chapter presents the automated workflow scripts that utilize the worker scripts introduced in the previous chapter. The pipeline structure is built using a python library called luigi, that was developed at Spotify. Luigi is a system, that can be used to control and monitor successive tasks. Because luigi was built for a music playing system that has hundreds of millions of users, it was deemed to be useful for handling the processing of genomic datasets, that consist of dozens if not hundreds of datafiles. Luigi works by

arranging tasks that must be done into a pipeline, that then successively moves from completed tasks to pending tasks. [27.]

The following luigi workflows (automated pipelines) introduced in chapters 4.5.1-3 are developed with object-oriented programming (OOP) in mind. They are built using the self-sufficient scripts as blocks or modules, and the pipeline running in the luigi workflow makes calls to these scripts whenever it requires one of them in the tasks.

4.5.1 TransferTracking.py

TransferTracking.py is the simplest luigi workflow in this group, as its only job is to run the monitor.py script (datasetlogger.py is embedded within monitor.py). This workflow can be seen as the first large circle in Figure 4 (Appendix 1). Its operation is automated using a crontab timer function in the cloud environment, but it can also be run manually. To understand the operation of this workflow, it's recommended to get acquainted with Chapter 4.4.1 and the associated Figure 7. The source file for TransferTracking.py is presented on Appendix 13.

Due to the nature of genomic datasets (consisting of multiple datafiles), this workflow can be parallelized over several supercomputers. Genomic datasets typically use terabytes of storage, so to process them efficiently the work is divided into smaller bunches. This workflow can be run to check all of the datafiles that make up a dataset, or a given fraction. The parallelization works by giving the script a number as a parameter that represents the complete dataset divided into that many parts. The command to initiate a run of this workflow is presented below on Example code 9, and the explanation of branches is given under the code.

```
luigi --module lega_mirroring.workflows.TransferTracking Check-
FilesInDirectory --branches 4 --branch 1 --config config.ini
```

Example code 9. Shell command to initiate a run of TransferTracking.py with given parallelization parameters (branches).

Branches indicate the fraction that is desired to be run, and branch is the section where the process begins from. When the workflow is given the parameters; branches: 4 and branch: 1, it means that one fourth (1 of 4, 25%) of datafiles are processed starting from file 1. Then the parallelization function will begin the process by generating a list of

contents in the transfer end-point directory taking only 25% of the files into account. The script takes the first file, then skips files 2, 3 and 4, because the parameter of branch number was chosen to be 1. Then the next file selected is file number 5, and after that, files number 6, 7 and 8 are skipped. In this way the process selects files according to Formula 1 below.

$$file\ number = branch + branches * n \tag{1}$$

So when branch number is 1 and branches are 4, the iterative process (n) generates a list of files with index numbers: 1, 5, 9, 13, 17, 21 and so on. Full file coverage is attained by launching consecutive runs on different machines with a varying branch number. So to process a directory with work divided over four computers, the used commands are as follows in Example code 10 below.

```
luigi --module lega_mirroring.workflows.TransferTracking Check-
FilesInDirectory --branches 4 --branch 1 --config config.ini

luigi --module lega_mirroring.workflows.TransferTracking Check-
FilesInDirectory --branches 4 --branch 2 --config config.ini

luigi --module lega_mirroring.workflows.TransferTracking Check-
FilesInDirectory --branches 4 --branch 3 --config config.ini

luigi --module lega_mirroring.workflows.TransferTracking Check-
FilesInDirectory --branches 4 --branch 4 --config config.ini
```

Example code 10. Shell commands to initiate four simultaneous runs of TransferTracking.py with work divided over four different computers.

The following four commands will process the contents of the directory in a way as presented numerically in Formula 2 below.

$$\begin{aligned} branch\ 1\ files &= 1, 5, 9, 13, 21, 25, 29, 33, 37 \\ branch\ 2\ files &= 2, 6, 10, 14, 22, 26, 30, 34, 38 \\ branch\ 3\ files &= 3, 7, 11, 15, 23, 27, 31, 35, 39 \\ branch\ 4\ files &= 4, 8, 12, 16, 24, 28, 32, 36, 40 \end{aligned} \tag{2}$$

As shown by the index numbers in Formula 2, calculated according to Formula 1, full coverage can be achieved with zero overlapping, resulting in highly improved processing

times, because the work is divided over several computers. The process can also be run as a single branch run by giving the parameters branches: 1 and branch: 1. The same parallelization is used in ProcessMaster.py, so the parallelization theory will not be repeated in the next chapter.

4.5.2 ProcessMaster.py

ProcessMaster.py is a type of preliminary step before files are taken into processing by TransferProcessing.py (see Figure 4/Appendix 1). Its main operation is to automate and handle sub-workflows of the actual processing scripts, and it can also be used to parallelize processing workflows as described in the previous chapter of 4.5.1 TransferTracking.py. Full source code for ProcessMaster.py is available on Appendix 15.

TransferProcessing.py is the most CPU-intensive (Central Processing Unit) process in the pipeline, as it contains encryption and checksum calculations. Therefore it was necessary to be able to parallelize the processes similarly to TransferTracking.py. Example code 11 below presents the shell command to initiate ProcessMaster.py.

```
luigi --module lega_mirroring.workflows.ProcessMaster Launch --
branches 4 --branch 1 --config config.ini
```

Example code 11. Shell command to initiate TransferProcessing.py automator (ProcessMaster.py) with given parallelization parameters.

The *Launch* class in Appendix 15 works by taking a list of contents in the processing directory from a preliminary function *par*, and then *yield*ing sub-workflows (TransferProcessing.py) for each file that has to be processed. An example process tree is presented below in Figure 7. The process tree's top-level task is the Launch class of ProcessMaster.py, which was given in the luigi shell command. The Launch class then branches out to as many TransferProcessing.py branches as were given in the listing parameters. Figure 8 is further discussed in the next chapter of 4.5.3 TransferProcessing.py.

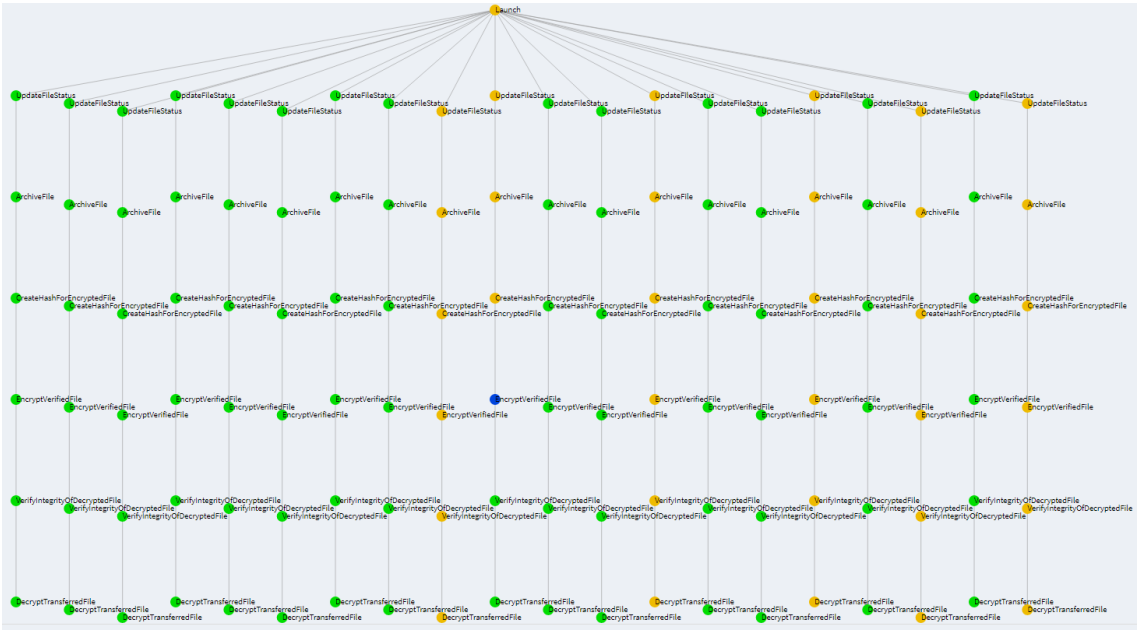


Figure 8. Example tasked workflow visualization from luigi web interface.

4.5.3 TransferProcessing.py

TransferProcessing.py is the actual data handler and archiver workflow in the pipeline. It consists of six tasks presented in the branches of Figure 8. TransferProcessing.py can also be seen in Figure 4 (Appendix 1) where it's located under ProcessMaster.py, as the right-most black circle. Full source code for TransferProcessing.py is available of Appendix 16.

In normal situations TransferProcessing.py is part of ProcessMaster.py, because it's easy to automate the creation of data processing workflows for each datafile. TransferProcessing.py can however be launched alone for a single selected file if needed, for example in problem situations to process one datafile that has been left behind from the dataset. Example code 12 below presents the shell command to initiate the data processing and archiving workflows of TransferProcessing.py. Note that this shell command doesn't have parallelization capabilities, as it's intended to be used on one file only.

```
luigi --module lega_mirroring.workflows.TransferProcessing
ArchiveFile --file /directory/processing/genomicdatafile.bam.gpg
--config config.ini
```

Example code 12. Shell command to initiate a single run of TransferProcessing.py targeted to a specific file.

When TransferProcessing.py is launched, it will take the selected file through six predetermined tasks; *DecryptTransferredFile, VerifyIntegrityOfDecryptedFile, EncryptVerifiedFile, CreateHashForEncryptedFile, ArchiveFile, UpdateFileStatus* as presented in Figure 9 below. The names of the tasks are descriptive, and each of them use a script from Chapter 4.4.

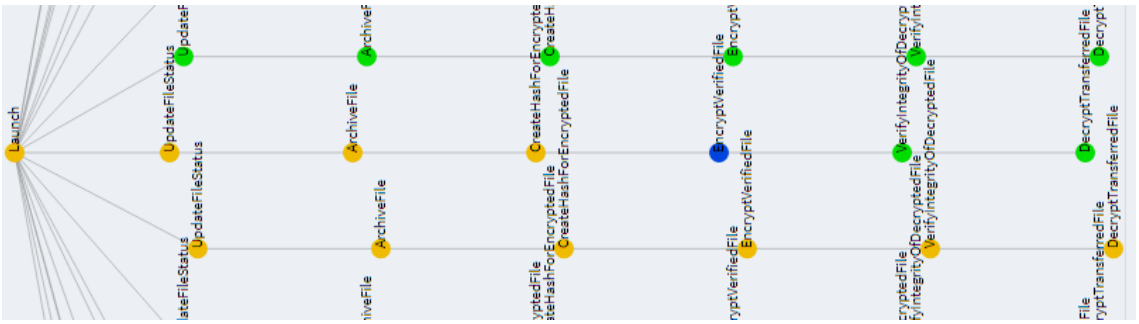


Figure 9. Cropped version of Figure 8, close-up view of the tasks of TransferProcessing.py.

The pipeline feature of luigi creates a fluid and automatic processing line, that will start handling the file from the first given task, that is the right-most circle in Figure 8. When this bottom-level task has been completed, luigi will move on to the next task following the line to the next circle. In the luigi interface (Figures 8 and 9), green circles are tasks that have been completed, yellow circles are pending tasks and blue circles represent tasks that are currently being worked on by the scripts.

4.6 Testing and Evaluation

Before compiling the script components into a uniform workflow, they were tested one by one to make sure they executed the intended operation correctly, and then again in an end-to-end test when the software module was completed. This test methodology ensures that the complete software package works as intended from start to finish. A set of four testing scripts were created to verify the operation of the data processing pipeline in an end-to-end test. The testing scripts are introduced below in Figure 10, and described below the figure.

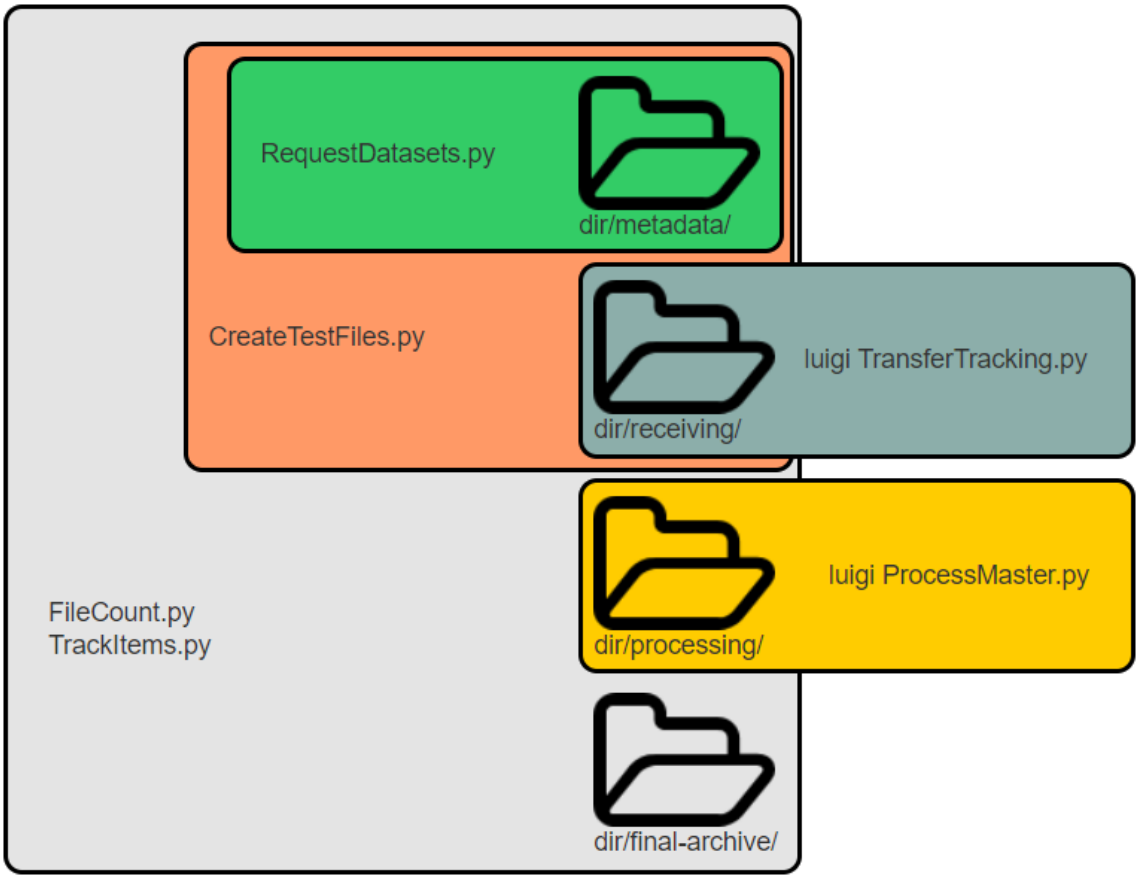


Figure 10. Testing directories with associated testing scripts and luigi workflows.

The end-to-end mock-up test begins from the red area with CreateTestFiles.py (Appendix 17). CreateTestFiles.py generates AES256 encrypted .bam files (genomic datafile file extension) into the *receiving* directory and .json dataset metadata files into the *metadata* directory. The .bam files are filled with lorem ipsum, but the metadata files follow the real metadata format that is requested from the EGA API. In a true scenario the metadata is directly inserted into the tracking database with information fetched from the EGA API.

When the test files have been created, TrackItems.py (Appendix 18) can be used to check the start and end conditions of the test. TrackItems.py is a multifunctional script with three parameters; start, end and compare. Start parameter is given after the test files have been created and before luigi workflows are run. This step generates a file that lists the contents of the receiving directory. After the test has been completed the other two parameters are given; end generates a similar file that lists the contents of the final-archive directory and compare then finds any differences between the two lists and reports them in a report file. The file can be viewed to see if any files were lost in the

process. An empty file indicates that the start and end directories are identical, so the process worked as intended.

RequestDatasets.py (Appendix 19) is another preliminary test file before the luigi workflows can be started. It works by reading the metadata .json files from the metadata directory, and inserting the datafile details into the tracking database. After this the luigi workflows can be run, starting with TransferTracking.py. A simple monitoring script called FileCount.py (Appendix 20) was also utilized in checking the amount of files in different directories, giving some indications to the progress of the data processing workload.

Several mock-up tests were carried out, and the largest test was done using 10 000 test files amounting to some 55 GB of storage. The test took roughly 22 minutes to complete, and proved that the software package is ready to be utilized on the production machine. Shorter demonstrations with fewer files and less storage were held for project partners and stakeholders in CSC during ELIXIR meetings.

The developed software scripts were praised for their clarity of expression and thorough github documentation and version control. A quick-guide was also written as a kind of cheat sheet, that explains the operation of the modules in detail, much like disclosed in the operation descriptions of this thesis. This quick-guide was distributed to the testers of the data processing pipeline, but it’s not discoverable through search engines. Because the quick-guide contains a convenient *foursquare* cheat sheet, its source link should be mentioned here for those that are interested. [28, p. 5.]

5 Conclusion

The aim of this thesis was to create a scalable and modular data processing pipeline. Python was chosen to be the programming language to create this software package due to it being lightweight and exhibiting good readability. PostgreSQL was the database system used in the production machine, but MySQL was chosen for the development phase, as it was easy to set up and maintain. MySQL and PostgreSQL fortunately resemble each other well, so transition from the development machine to the production machine required few alterations. The thesis progressed steadily, switching between design and development phases daily as problems occurred. Python's RAD feature was great in this sense, as the scripts required no compiling, and testing new code was quick and easy.

The scripts created in this thesis will become part of the European ELIXIR bioinformatics infrastructure. The program is implemented in the receiving end of each local EGA in the ELIXIR nodes where it processes and archives transmitted genomic datasets. Due to the modularity of the created scripts, the operation of the program can be altered with relatively low effort. This was one major desire for the modules, so that they could be repurposed into processing and archiving other kinds of data as well; e.g. old newspapers and medical images – in fact, the created data processing pipeline is currently being investigated to be used in a *digipathology* project that aims to digitize and archive millions of tissue samples (by taking pictures of physical biological samples) from Finnish biobanks.

The project was conducted according to schedule, and the supervisors to the thesis were pleased with the pace and quality of work. Daily meetings would be held with instructors and weekly meetings with other project stakeholders. A great deal was learned in this thesis related to software development and bioinformatics. New skills were also attained, that will prove to be useful in future design and development projects. Due to learning many new things and improving one's programming skills, creating a second, improved version of the tasked workflow would surely simplify the overall procedures of the program. "*Now that I am wiser"*.

References

1 Ayme, Ségolène et al. 2004. Analysis of genomic research supported under FP5. Online Material. <http://ec.europa.eu/smart-regulation/evaluation/search/download.do?documentId=2247> Read on: 31.1.2018.

2 Pohlhaus, Jennifer & Cook-Deegan, Robert. 2008. Genomics Research: World Survey of Public Funding. Online Material. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2576262/> Read on: 31.1.2018.

3 ELIXIR. 2014. About us. Online Material. <https://www.elixir-europe.org/about-us> Read on: 31.1.2018.

4 CSC – IT Center for Science. 2016. CSC – Finnish expertise in ICT for research, education, culture and public administration. Online Material. <https://www.csc.fi/csc> Read on: 31.1.2018.

5 European Commission. 2014. Horizon 2020 in brief. Online Material. <http://ec.europa.eu/programmes/horizon2020/sites/horizon2020/files/H2020_inBrief_EN_FinalBAT.pdf> Read on: 1.2.2018.

6 Blomberg, Niklas. 2015. Introduction to ELIXIR-EXCELERATE. Online Material. <https://www.elixir-europe.org/system/files/documents/excelerate-introduction.pdf> Read on: 1.2.2018.

7 Freeworldmaps. Europe Blank Map. Online Material. Modified by Author. <http://www.freeworldmaps.net/europe/blank_map.html> Read on: 1.2.2018.

8 ELIXIR. 2016. EXCELERATE WP9 – An ELIXIR framework for secure archiving, dissemination and analysis of human access-controlled data; enabling biobanks, cohorts and local resource services to leverage the EGA. Online Material. <https://drive.google.com/file/d/0B4WQQq4hwmbQcEV5SkhtN1BYMTQ/view> Read on: 1.2.2018.

9 EMBL-EBI. About us. Online Material. <https://www.ebi.ac.uk/about> Read on: 1.2.2018.

10 Center for Genomic Regulation. Mission, vision and values. Online Material. <http://www.crg.eu/node/73> Read on: 1.2.2018.

11 Institute for Research in Biomedicine. 2014. Spain to become a full member of ELIXIR, the European infrastructure for bioinformatics. Online Material. <https://www.irbbarcelona.org/en/news/spain-to-become-a-full-member-of-elixir-the-european-infrastructure-for-bioinformatics> Read on: 1.2.2018.

12 NeIC. What is NeIC?. Online Material. <https://neic.no/about/> Read on: 1.2.2018.

13 Lister Hill National Center for Biomedical Communications. 2018. Help Me Understand Genetics: The Human Genome Project. Online Material. <https://ghr.nlm.nih.gov/primer/hgp.pdf> Read on: 2.2.2018.

14 EMBL-EBI. 2017. Homo Sapiens Whole Genome. Online Material. <https://www.ensembl.org/Homo_sapiens/Location/Genome> Read on: 2.2.2018.

15 Lappalainen, Ilkka. 2017. Deputy Head of ELIXIR Finland, CSC, Espoo. Orientation to ELIXIR and Bioinformatics. Discussions at CSC.

16 Chacon, Scott & Straub, Ben. 2014. Pro Git 2nd ed. Online Material. <https://github.com/progit/progit2/releases/download/2.1.35/progit.pdf> Read on: 6.2.2018.

17 Kataja, Teemu. 2017. Local EGA: Data Mirroring. Online Material. <https://github.com/CSCfi/lega-mirroring> Read on: 6.2.2018.

18 Python. What is Python? Executive Summary. Online Material. <https://www.python.org/doc/essays/blurb/> Read on: 6.2.2018.

19 Beal, Vangie. High-level language. Online Material. <https://www.webopedia.com/TERM/H/high_level_language.html> Read on: 6.2.2018.

20 Tutorials Point Ltd. 2016. MySQL Database Management System. Online Material. <https://www.tutorialspoint.com/mysql/mysql_tutorial.pdf> Read on: 6.2.2018.

21 AXURE. Features. Online Material. <https://www.axure.com/#a=features> Read on: 6.2.2018.

22 YatriTrivedi. 2015. The Beginner's Guide to Shell Scripting: The Basics. Online Material. <https://www.howtogeek.com/67469/the-beginners-guide-to-shell-scripting-the-basics/> Read on: 6.2.2018.

23 CSC. ePouta IaaS Cloud. Online Material. <https://research.csc.fi/epouta> Read on: 6.2.2018.

24 Zhu, Hong. 2005. Software Design Methodology: From Principles to Architectural Styles. UK: Elsevier Science.

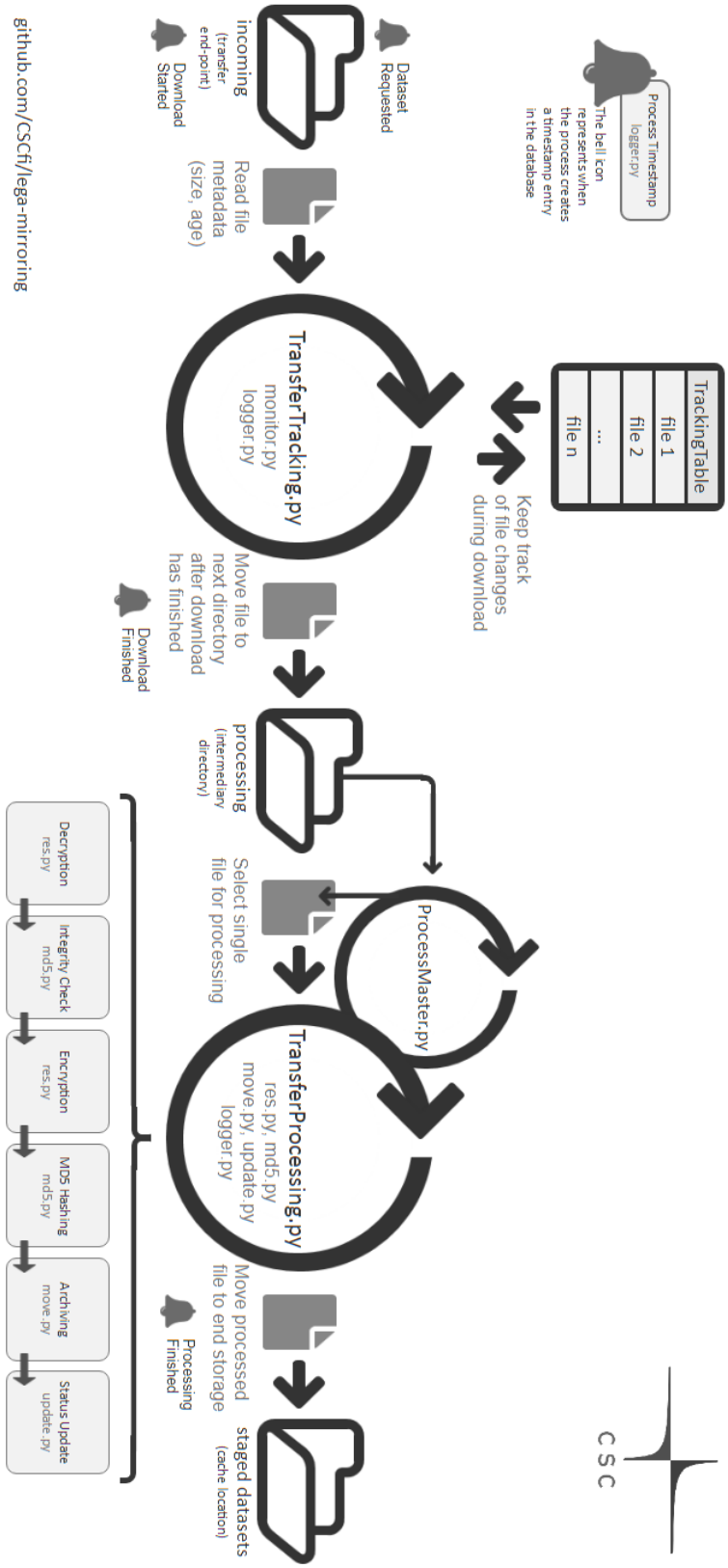
25 Senf, Alexander. 2018. Re/Encryption Provider Service. Online Material. <https://github.com/elixir-europe/ega-data-api-v3-res_mvc> Read on: 13.2.2018.

26 Rouse, Margaret. 2005. MD5. Online Material. <http://searchsecurity.techtarget.com/definition/MD5> Read on: 13.2.2018.

27 Bernhardsson, Erik & Freider, Elias. 2012. Luigi. Online Material. <https://github.com/spotify/luigi> Read on: 15.2.2018.

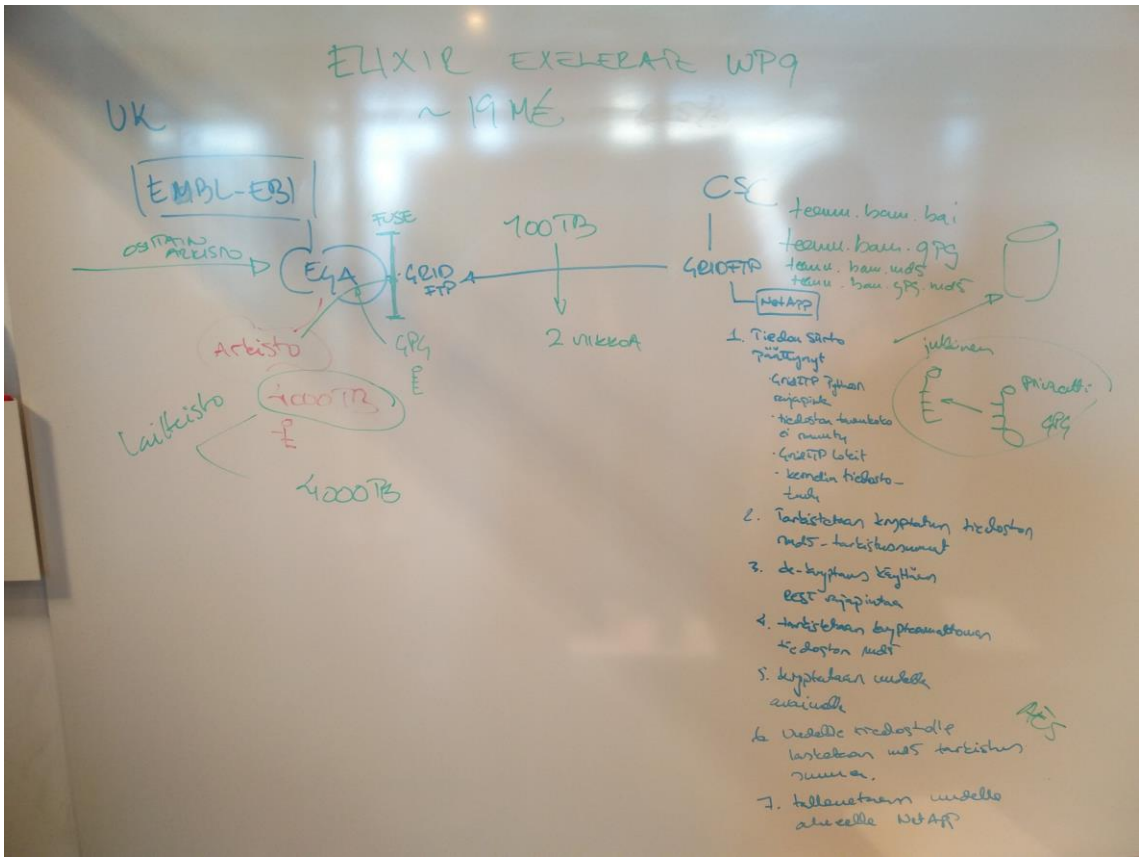
28 Kataja, Teemu. 2017. Local EGA Data Mirroring Prototype Guide. Online Material. <https://docs.google.com/document/d/1gl3Ob3GBkx5JE0QSAD-Wu9qtnjROX4u1JA8tfvcAr0s/edit?usp=sharing> Read on: 2.3.2018.

Enlarged version of Figure 4

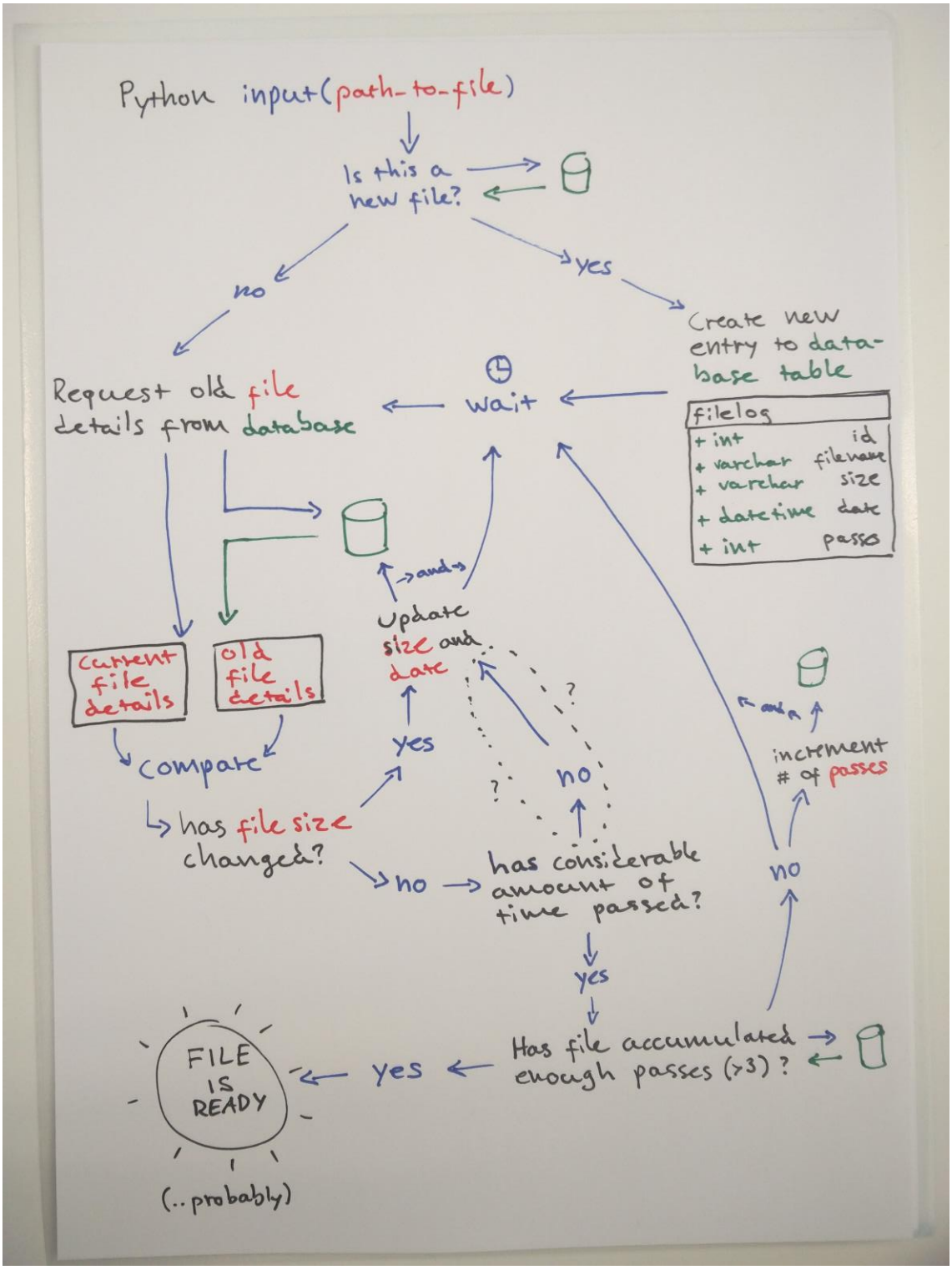


Initial software requirement specifications

Initial software requirement specifications given at CSC before starting the design and development phase of the software. [15.]



First draft for the operation of monitor.py



Database creation script

Source file for database creation script is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/other/db_script.sql>.

```
CREATE DATABASE dev_ega_downloader;
USE dev_ega_downloader;


# For TransferTracking.py
CREATE TABLE trackingtable (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(128) NOT NULL,
    size VARCHAR(32) NOT NULL,
    age VARCHAR(32) NOT NULL,
    passes INT NOT NULL,
    PRIMARY KEY (id)
);


# For TransferProcessing.py
CREATE TABLE file (
    file_id VARCHAR(128) NULL,
    file_name VARCHAR(128) NULL,
    file_size INT8 NULL,
    file_md5 VARCHAR(32) NULL,
    status VARCHAR(13) NULL
);


CREATE UNIQUE INDEX file_id_idx ON dev_ega_downloader.file (file_id);


CREATE TABLE filedataset (
    dataset_id VARCHAR(128) NULL,
    file_id VARCHAR(128) NULL
);


CREATE UNIQUE INDEX dataset_id_idx ON dev_ega_downloader.filedataset (da-
taset_id, file_id);
CREATE UNIQUE INDEX file_id_idx ON dev_ega_downloader.filedataset (file_id,
dataset_id);


# Dataset logger
CREATE TABLE dataset_log (
    dataset_id VARCHAR(128),
    n_files INT,
    n_bytes INT,
    date_requested TIMESTAMP NULL,
    date_download_start TIMESTAMP NULL,
    date_download_end TIMESTAMP NULL,
    date_processing_end TIMESTAMP NULL
);


CREATE UNIQUE INDEX dataset_id_idx ON dev_ega_downloader.dataset_log (da-
taset_id);
```

Configuration file

The following contents make up the configuration file *config.ini*, which is used to configure the scripts to work properly. Without the configuration file and correct values for the variables the scripts will not work. Note that some variables are URLs to API services that must be installed beforehand. Source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/config.ini>.

```
[database]
# localhost or url to mysql server
host=localhost
# login username to your database
user=root
# password to to your database with given username
passwd=root
# database to use
db=dev_ega_downloader
[func_conf]
# chunk size in bytes used in some functions
chunk_size=16384
# number of seconds before monitor.py starts marking passes
age_limit=300
# number of passes required for monitor.py to determine that download has
finished
pass_limit=3
# url to RES microservice
res_url=http://localhost:9090/file/
# list of encryption extensions allowed (.bam is unencrypted)
extensions=.cip,.gpg
[api]
# EGA API address
api_url=https://ega.ebi.ac.uk:8051/elixir/central/basic/DATASETID/manifest
# username to access EGA API
api_user=appuser
# password for api_user
api_pass=9thwfosdg3ou4otygohbl
[workspaces]
# the incoming directory, where downloads are received
receiving=/data/incoming/gridftp-endpoint/
# the processing directory, intermediary
processing=/data/incoming/processing/
# the cache location, where completed files are stored
end_storage=/data/incoming/final-archive/
# testing directory for EGA API mockups
metadata=/data/incoming/metadata/
```

monitor.py

monitor.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/scripts/monitor.py>.

```
#!/usr/bin/env python3.4


import pymysql
import os
import time
import datetime
import calendar
import sys
import argparse
import logging
from configparser import ConfigParser
from collections import namedtuple
import lega_mirroring.scripts.datasetlogger


# Log events to file
logging.basicConfig(filename='monitor_log.log',
                    format='%(asctime)s %(message)s',
                    datefmt='%d-%m-%Y %H:%M:%S',
                    level=logging.INFO)



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'chunk': config.getint('func_conf', 'chunk_size'),
            'age_limit': config.getint('func_conf', 'age_limit'),
            'pass_limit': config.getint('func_conf', 'pass_limit'),
            'path_receiving': config.get('workspaces', 'receiving'),
            'path_processing': config.get('workspaces', 'processing')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object
    :hostname: address of mysql server
    :username: username to log in to mysql server
```

```
    :password: password associated with :username: to log in to mysql
server
    :database: database to be worked on
    """
    db = pymysql.connect(host=hostname,
                         user=username,
                         passwd=password,
                         db=database)
    return db



def get_file_size(path):
    """
    This function reads a file and returns it's byte size as numeral string
    :path: file to be read
    """
    return os.path.getsize(path)



def get_file_age(path):
    """
    This function reads a file and returns it's last modified date as
    mtime(float) in string form
    :path: file to be read
    """
    return os.path.getmtime(path)



def get_time_now():
    """
    This function returns the current time as mtime(float) in string form
    """
    return calendar.timegm(time.gmtime())



def db_get_file_details(path, db):
    """
    This function queries the database for file details
    and returns a list of results or false
    :path: filename to be queried
    :db: database connection object
    """
    status = False
    cur = db.cursor()
    cur.execute('SELECT * '
                'FROM trackingtable '
                'WHERE name=%s;',
                [path])
    result = cur.fetchall()
    if cur.rowcount >= 1:
        for row in result:
            status = {'id': row[0],
                      'name': path,
                      'size': int(row[2]),
                      'age': float(row[3]),
                      'passes': row[4]}
```

```
    return status



def db_update_file_details(path, db):
    """
    This function updates file size and age to database
    as well as resets the passes value to zero
    :path: filename
    :db: database connection object
    """
    file_size = get_file_size(path)
    file_age = get_file_age(path)
    file_id = db_get_file_details(path, db)['id']
    params = [file_size, file_age, file_id]
    cur = db.cursor()
    cur.execute('UPDATE trackingtable '
                'SET size=%s, '
                'age=%s, '
                'passes=0 '
                'WHERE id=%s;',
                params)
    db.commit()
    return



def db_increment_passes(path, db):
    """
    This function increments the number of passes by 1
    :path: filename
    :db: database connection object
    """
    file_id = db_get_file_details(path, db)['id']
    file_passes = db_get_file_details(path, db)['passes']+1
    params = [file_passes, file_id]
    cur = db.cursor()
    cur.execute('UPDATE trackingtable '
                'SET passes=%s '
                'WHERE id=%s;',
                params)
    db.commit()
    return



def db_insert_new_file(path, db):
    """
    This function creates a new database entry of a new file
    :path: filename
    :db: database connection object
    """
    file_size = get_file_size(path)
    file_age = get_file_age(path)
    params = [path, file_size, file_age]
    cur = db.cursor()
    cur.execute('INSERT INTO trackingtable '
                'VALUES (NULL, %s, %s, %s, 0);',
                params)
    db.commit()
    return
```

```
def log_event(path, db):
    """
    This function prints monitoring events to log file
    :path: filename
    :db: database connection object
    """
    time_now = get_time_now()
    file_size = db_get_file_details(path, db)['size']
    file_age = db_get_file_details(path, db)['age']
    file_passes = db_get_file_details(path, db)['passes']
    logging.info(path + ' last updated: ' + str(file_age) +
                 ' size: ' + str(file_size) + ' passes: ' +
str(file_passes))
    return



def par(branches, branch, pathr):
    """
    This function reads a directory and generates a list
    of files to be checked by a single parallel process
    :branches:  number of branches to be run
    :branch:  id of branch
    :pathr:  path_receiving from config.ini
    """
    complete_set = []  # to be appended
    selected_set = []  # to be appended
    for root, dirs, files in os.walk(pathr):
        for item in files:
            # form full path
            fullpath = os.path.join(root, item)
            # strip path (to leave subdirs if they exist)
            relpath = fullpath.replace(pathr, '')
            complete_set.append(relpath)
    i = 0
    while i <= len(complete_set):
        index = branch+i*branches
        if index <= len(complete_set):
            selected_set.append(complete_set[index-1])
        i += 1
    return selected_set



def lookup_dataset_id(db, file):
    """
    This function finds the dataset id the given file
    belongs to and returns it as a string
    :db: database connection object
    :file: datafile belonging to a dataset
    """
    dataset_id = 0
    cur = db.cursor()
    cur.execute('SELECT dataset_id '
                'FROM filedataset '
                'WHERE file_id=('
                'SELECT file_id '
                'FROM file '
                'WHERE file_name=%s);',
                [file])
```

```
    result = cur.fetchall()
    if cur.rowcount >= 1:
        for row in result:
            dataset_id = row[0]
    return dataset_id


'''****************************************************************'''
#                          cmd-executable                          #
'''****************************************************************'''



def main(arguments=None):
    """
    This function runs the script
    :arguments: contains parsed command line parameters
    """
    start_time = time.time()
    args = parse_arguments(arguments)
    branches = int(args.branches)
    branch = int(args.branch)
    conf = args.config
    # Get configuration values from external file
    config = get_conf(conf)
    path = config.path_receiving
    # Establish database connection
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    # Begin file checking process
    selected_set = par(branches, branch, config.path_receiving)
    for file in selected_set:
        rawfile = file
        file = os.path.join(path, file)
        if db_get_file_details(file, db):  # Old file
            if db_get_file_details(file, db)['passes'] < config.pass_limit:
                # File transfer is incomplete
                if (get_file_size(file) >
                        db_get_file_details(file, db)['size']):
                    #   File size has changed
                    db_update_file_details(file, db)
                else:  # File size hasn't changed
                    if (get_time_now() -
                            db_get_file_details(file, db)['age'] >
                            config.age_limit):
                        #   File hasn't changed in some time
                        # Mark a pass on db table 0..3
                        db_increment_passes(file, db)
                log_event(file, db)
            else:
                # move file from receiving(wf1) to processing(wf2)
                try:
                    os.rename(file, os.path.join(config.path_processing,
                              rawfile))
                    # put timestamp to dataset_log table
                    dataset_id = lookup_dataset_id(db, file)
                    lega_mirroring.scripts.datasetlogger.main(
                        ['date_download_end', dataset_id, conf])
                except:
                    pass
        else:  # New file
            # Create new directory to processing
```

```
            if os.path.dirname(rawfile):
                if not os.path.exists(os.path.join(config.path_processing,
                                                   os.path.dirname(raw-
file))):
                    os.mkdir(os.path.join(config.path_processing,
                                          os.path.dirname(rawfile)))
            db_insert_new_file(file, db)
            # put timestamp to dataset_log table
            dataset_id = lookup_dataset_id(db, file)
            lega_mirroring.scripts.datasetlogger.main(['date_down-
load_start',
                                                       dataset_id, conf])
            log_event(file, db)
    return ('Runtime: ' + str(time.time()-start_time) + ' seconds')



def parse_arguments(arguments):
    """
    This function parses command line inputs and returns them for main()
    :branches: this is the total number of parallelizations to be run
    :branch: this is a fraction of the parallelizations, e.g. 1 of 4
    :config: full path to config.ini (or just config.ini if
             cwd: lega-mirroring)
    """
    parser = argparse.ArgumentParser(description='Check files\' age and
size'
                                     ' in target directory and track them '
                                     ' using a MySQL database.')
    parser.add_argument('branches',
                        help='number of parallelizations')
    parser.add_argument('branch',
                        help='unique id of machine')
    parser.add_argument('config',
                        help='location of configuration file')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

Res.py

res.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/scripts/res.py>.

```
#!/usr/bin/env python3.4


import requests
import sys
import os
import argparse
import logging
from configparser import ConfigParser
from collections import namedtuple


logging.basicConfig(filename='res_log.log',
                    format='%(asctime)s %(message)s',
                    datefmt='%d-%m-%Y %H:%M:%S',
                    level=logging.INFO)



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'chunk_size': config.getint('func_conf', 'chunk_size'),
            'res_url': config.get('func_conf', 'res_url'),
            'extensions': config.get('func_conf', 'extensions')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def decrypt(host_url, file_path):
    """
    This function sends an HTTP request to an active
    RES microservice and returns a stream of decrypted data
    :host_url: address of RES microservice
    :file_path: full path to .cip file to be decrypted
    """
    params = {'filePath': file_path,
              'sourceFormat': 'aes128',
              'sourceKey': 'aeskey',
              'destinationFormat': 'plain'}
    r = requests.get(host_url, params, stream=True)
    if not r:
        raise Exception('decryption failed')
    log_event(r, host_url, file_path)
    return r
```

```
def encrypt(host_url, file_path):
    """
    This function sends an HTTP request to an active
    RES microservice and returns a stream of encrypted data
    :host_url: address of RES microservice
    :file_path: full path to .bam file to be encrypted
    """
    params = {'filePath': file_path,
              'destinationFormat': 'aes128',
              'destinationKey': 'aeskey'}
    r = requests.get(host_url, params, stream=True)
    if not r:
        raise Exception('encryption failed')
    log_event(r, host_url, file_path)
    return r



def write_to_file(crypt, feed, chnk, path, ext):
    """
    This function handles a stream of data and writes
    it to file. The function determines the file extension
    by itself according to the given method (main parameter)
    :crypt: operating method given in main
    :feed: stream of decrypted/encrypted data
    :chnk: size of data read from stream and written to file
    :path: full path to destination file
    """
    if crypt == 'decrypt':
        newpath = path  # .bam
        if path.endswith(tuple(ext)):
            newpath, extension = os.path.splitext(path)
        with open(newpath, 'wb+') as f:
            for chunk in feed.iter_content(chunk_size=chnk):
                if chunk:
                    f.write(chunk)
    elif crypt == 'encrypt':
        newpath = path + '.cip.csc'
        with open(newpath, 'wb+') as f:
            for chunk in feed.iter_content(chunk_size=chnk):
                if chunk:
                    f.write(chunk)
    else:
        raise Exception('invalid var(crypt) for write_to_file()')
    return



def log_event(event, host, path):
    """
    This function logs successes and failures to file
    :event: pass or failed
    :host: address of RES microservice
    :path: full path to original file (before crypt-operations)
    """
    if event:
        logging.info(' OK: http-request: ' + host +
                     ' path: ' + path)
    else:
```

```
        logging.info(' ERROR: Check that url and path are correct -'
                     ' http-request: ' + host +
                     ' path: ' + path)
    return



'''****************************************************************'''
#                           cmd-executable                         #
'''****************************************************************'''



def main(arguments=None):
    """
    This function runs the script
    :arguments: contains parsed command line parameters
    """
    args = parse_arguments(arguments)
    method = args.method
    path = args.path
    conf = args.config
    config = get_conf(conf)
    ext = tuple(config.extensions.split(','))
    if method == 'decrypt':
        write_to_file(method, decrypt(config.res_url, path),
                      config.chunk_size, path, ext)
    elif method == 'encrypt':
        write_to_file(method, encrypt(config.res_url, path),
                      config.chunk_size, path, ext)
    else:
        raise Exception('invalid method, must be \'encrypt\' or \'de-
crypt\'')
    return



def parse_arguments(arguments):
    """
    This function parses command line inputs and returns them for main()
    :method: parameter that determines the operation of the script
             either encrypt or decrypt, can not be left empty
    :path: path to file to be worked on
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    parser = argparse.ArgumentParser(description='Utilizes RES Micro-
service'
                                     ' to decrypt or encrypt files.')
    parser.add_argument('method',
                        help='encrypt or decrypt')
    parser.add_argument('path',
                        help='path to file to be worked on')
    parser.add_argument('config',
                        help='location of config.ini')
    return parser.parse_args(arguments)



if __name__ == '__main__':
```

```
RETVAL = main()
sys.exit(RETVAL)
```

md5.py

md5.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/scripts/md5.py>.

```
#!/usr/bin/env python3.4


import pymysql
import sys
import argparse
import logging
import hashlib
import os
from configparser import ConfigParser
from collections import namedtuple


logging.basicConfig(filename='md5_log.log',
                    format='%(asctime)s %(message)s',
                    datefmt='%d-%M-%Y %H:%M:%S',
                    level=logging.INFO)



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'chunk_size': config.getint('func_conf', 'chunk_size'),
            'extensions': config.get('func_conf', 'extensions'),
            'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'path_gridftp': config.get('workspaces', 'receiving'),
            'path_processing': config.get('workspaces', 'processing')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object
    :hostname: address of mysql server
    :username: username to log in to mysql server
    :password: password associated with :username: to log in to mysql
server
    :database: database to be worked on
    """
    db = pymysql.connect(host=hostname,
```

```
                        user=username,
                        passwd=password,
                        db=database)
    return db



def hash_md5_for_file(method, path, chunk_size, ext):
    """
    This function reads a file and returns a generated md5 checksum
    :method: operating method given in main, hash or check
             hash: md5 is generated and written to .md5 file
             check: md5 is generated and returned
    :path: path to file that md5 will be hashed for
    :chunk_size: chunk size for reading original file
    """
    hash_md5 = hashlib.md5()
    md5 = False
    if os.path.exists(path):
        with open(path, 'rb') as f:
            for chunk in iter(lambda: f.read(chunk_size), b''):
                hash_md5.update(chunk)
            md5 = hash_md5.hexdigest()
            if method == 'hash':
                if not path.endswith(ext):
                    # if path is file.bam, add .cip.csc to it
                    path = path + '.cip.csc'
                with open(path + '.md5', 'w') as fmd5:
                    fmd5.write(md5)
    else:
        raise Exception('file ' + path + ' not found')
    return md5



def db_fetch_md5(db, path_file, path_gridftp, path_processing):
    """
    This function queries the database for an md5 hash matching
    the given filename and returns it
    :db: database connection object
    :path_file: path to file to be checked
    :path_gridftp: path to receiving folder, needed for db query
    """
    filename = path_file.replace(path_processing, path_gridftp)
    md5 = False
    cur = db.cursor()
    cur.execute('SELECT file_md5 '
                'FROM file '
                'WHERE file_name=%s;',
                [filename])
    result = cur.fetchall()
    if cur.rowcount >= 1:
        for row in result:
            md5 = row[0]
    return md5



'''*************************************************************'''
#                          cmd-executable                       #
'''*************************************************************'''
```

```
def main(arguments=None):
    """
    This function runs the script
    :arguments: contains parsed command line parameters
    """
    args = parse_arguments(arguments)
    config = get_conf(args.config)
    ext = tuple(config.extensions.split(','))
    # Establish database connection
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    retval = False
    # Generate md5 hash and save to file.md5
    if args.method == 'hash':
        md5 = hash_md5_for_file(args.method, args.path, config.chunk_size,
ext)
        retval = md5  # always true if path exists
        if md5:
            logging.info('Created md5 hash for ' + args.path +
                         ' (' + md5 + ')')
        else:
            logging.info('Error creating md5 hash for ' + args.path +
                         ', file not found.')
    # Read md5 checksum from database and compare it to hashed value
    elif args.method == 'check':
        md5 = hash_md5_for_file(args.method, args.path, config.chunk_size,
ext)
        key_md5 = db_fetch_md5(db, args.path, config.path_gridftp,
                               config.path_processing)
        retval = (md5 == key_md5)  # true if checksums match
        if md5 == key_md5:
            logging.info('OK (md5 checksums match)'
                         ' File: ' + args.path +
                         ' Hashed md5: ' + md5 +
                         ' Received md5: ' + str(key_md5))
        else:
            logging.info('ERROR (md5 checksums don\'t match)'
                         ' File: ' + args.path +
                         ' Hashed md5: ' + md5 +
                         ' Received md5: ' + str(key_md5))
    else:
        raise Exception('invalid method, but be \'hash\' or \'check\'')
    return retval



def parse_arguments(arguments):
    """
    This function parses command line inputs and returns them for main()
    :method: parameter that determines the operation of the script
             either hash or check, can not be left empty
    :path: path to file to be worked on
    :config: full path to config.ini (or just config.ini if
                    cwd: lega-mirroring)
    """
    parser = argparse.ArgumentParser(description='Generate md5 hash '
                                     'or check md5 sum '
                                     'for given file.')
    parser.add_argument('method',
```

```
                        help='hash or check.')
    parser.add_argument('path',
                        help='path to file that will be checked or
hashed.')
    parser.add_argument('config',
                        help='path to configuration file.')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

move.py

move.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/scripts/move.py>.

```
#!/usr/bin/env python3.4


import os
import argparse
import sys
import logging
from configparser import ConfigParser
from collections import namedtuple


logging.basicConfig(filename='move_log.log',
                    format='%(asctime)s %(message)s',
                    datefmt='%d-%m-%Y %H:%M:%S',
                    level=logging.INFO)



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'end_storage': config.get('workspaces', 'end_storage'),
            'path_processing': config.get('workspaces', 'processing')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def move(file, md5, dest, pathp):
    """
    This function moves files from current directory to another directory
    atomically, and removes the original working files
    :file: file that will be moved
    :md5: associated md5-file that will be moved
    :dest: destination directory
    """
    try:
        basefile = file.replace(pathp, '')
        basemd5 = md5.replace(pathp, '')
        os.rename(file, os.path.join(dest, basefile))
        os.rename(md5, os.path.join(dest, basemd5))
        # Remove up to two extensions
        #
        # THIS NEEDS SOME THINKING AND TESTING
        #                                       was (file)
        basefile, extension = os.path.splitext(basefile)
        basefile, extension = os.path.splitext(basefile)
```

```
        try:
            os.remove(os.path.join(pathp, basefile))  # rm .bam
            os.remove(os.path.join(pathp, basefile + '.cip'))  # rm
.bam.cip
        except:
            pass
    except:
        pass
    log_event(file, dest)
    return



def log_event(file, dest):
    """
    This function logs moving events
    :file: file that was moved
    :dest: destination directory of file
    """
    logging.info(file + ' moved to ' + dest)
    return



'''*************************************************************'''
#                          cmd-executable                       #
'''*************************************************************'''



def main(arguments=None):
    """
    This function runs the script
    :arguments: contains parsed command line parameters
    """
    args = parse_arguments(arguments)
    conf = args.path_to_config
    config = get_conf(conf)
    move(args.file, args.md5, config.end_storage, config.path_processing)
    return



def parse_arguments(arguments):
    """
    This function parses command line inputs and returns them for main()
    :file: file.bam.cip.csc that will be moved final-archive
    :md5: file.bam.cip.csc.md5 that will be moved to final-archive
    """
    parser = argparse.ArgumentParser(description='Move file to'
                                     ' predetermined location set in'
                                     ' config.ini')
    parser.add_argument('file',
                        help='path to file to be moved')
    parser.add_argument('md5',
                        help='path to md5 to be moved')
    parser.add_argument('path_to_config',
                        help='path to configuration file.')
    return parser.parse_args(arguments)
```

```
if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

update.py

update.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/scripts/update.py>.

```
#!/usr/bin/env python3.4


import pymysql
import sys
import argparse
import logging
import os
from configparser import ConfigParser
from collections import namedtuple
import lega_mirroring.scripts.datasetlogger


logging.basicConfig(filename='update_log.log',
                    format='%(asctime)s %(message)s',
                    datefmt='%d-%M-%Y %H:%M:%S',
                    level=logging.INFO)



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'path_archive': config.get('workspaces', 'end_storage'),
            'path_gridftp': config.get('workspaces', 'receiving')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object
    :hostname: address of mysql server
    :username: username to log in to mysql server
    :password: password associated with :username: to log in to mysql
server
    :database: database to be worked on
    """
    db = pymysql.connect(host=hostname,
                         user=username,
                         passwd=password,
```

```
                            db=database)
    return db



def db_update_file(db, filename, path_archive, path_gridftp):
    """
    This function updates file path and status to database
    :db: database connection object
    :path_file: path to file
    :path_archive: path to end storage
    :path_gridftp: path to receivings directory
    """
    oldfile = os.path.join(path_gridftp, filename)
    newfile = os.path.join(path_archive, filename)
    params = ['available', newfile, oldfile]
    cur = db.cursor()
    cur.execute('UPDATE file '
                'SET status=%s, file_name=%s '
                'WHERE file_name=%s;',
                params)
    db.commit()
    return



def lookup_dataset_id(db, file):
    """
    This function finds the dataset id the given file
    belongs to and returns it as a string
    :db: database connection object
    :file: datafile belonging to a dataset
    """
    dataset_id = 0
    cur = db.cursor()
    cur.execute('SELECT dataset_id '
                'FROM filedataset '
                'WHERE file_id=('
                'SELECT file_id '
                'FROM file '
                'WHERE file_name=%s);',
                [file])
    result = cur.fetchall()
    if cur.rowcount >= 1:
        for row in result:
            dataset_id = row[0]
    return dataset_id



'''*****************************************************************'''
#                           cmd-executable                          #
'''*****************************************************************'''



def main(arguments=None):
    """
    This function runs the script
    :arguments: contains parsed command line parameters
```

```
    """
    args = parse_arguments(arguments)
    conf = args.config
    # Get configuration values from external file
    config = get_conf(conf)
    # Establish database connection
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    db_update_file(db, args.path, config.path_archive, config.path_gridftp)
    # put timestamp to dataset_log table
    file = os.path.join(config.path_archive, args.path)
    dataset_id = lookup_dataset_id(db, file)
    lega_mirroring.scripts.datasetlogger.main(['date_processing_end',
                                               dataset_id, conf])
    return



def parse_arguments(arguments):
    """
    This function parses command line inputs and returns them for main()
    :path: path to file that\'s details are updated
    :config: full path to config.ini (or just config.ini if
             cwd: lega-mirroring)
    """
    parser = argparse.ArgumentParser(description='Update file status '
                                     'and path')
    parser.add_argument('path',
                        help='base filename')
    parser.add_argument('config',
                        help='path to configuration file.')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

dsin.py

dsin.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/other/dsin.py>.

```
#!/usr/bin/env python3.4


import pymysql
import sys
import argparse
import os
from configparser import ConfigParser
from collections import namedtuple


'''
    # dsin.py Dataset Input #

    This script is used to input metadata to dev_ega_downloader tables
    file, filedataset and dataset_log from external (non-EGA format)
sources.
'''



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object

    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'path_incoming': config.get('workspaces', 'receiving')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object

    :hostname: address of mysql server
    :username: username to log in to mysql server
    :password: password associated with :username: to log in to mysql
server
    :database: database to be worked on
    """
```

```
    db = pymysql.connect(host=hostname,
                         user=username,
                         passwd=password,
                         db=database)
    return db


def parse_file(file):
    """
    This function reads a given file and returns a list of rows

    :file: file input
    """
    rows = []
    for line in open(file, 'r'):
        rows.append(line.strip().split('\t'))
    return rows


def db_insert_metadata(db, dataset_id, metadata, path_incoming):
    """
    This function inserts received data to database

    :db: database connection object
    :dataset_id: string, e.g. EGAD000 or SN000
    :metadata: list of rows containing datafile metadata
    """
    cur = db.cursor()
    for i in range(len(metadata)):
        file_id = metadata[i][0].replace('.bam', '')
        file_name = metadata[i][0]
        file_name = os.path.join(path_incoming, file_name)
        file_md5 = metadata[i][1]
        params_file = [file_id, file_name, file_md5]
        params_data = [dataset_id, file_id]
        try:
            cur.execute('INSERT INTO file VALUES '
                        '(%s, %s, NULL, %s, "pending");', params_file)
            cur.execute('INSERT INTO filedataset VALUES '
                        '(%s, %s);', params_data)
        except:
            pass
        i += 1
    db.commit()
    return


def db_date_requested(db, dataset_id):
    """
    This function update the dataset_log table to fill in a requested date

    :db: database connection object
    :dataset_id: string, e.g. EGAD000 or SN000
    """
    cur = db.cursor()
    cur.execute('INSERT INTO dataset_log '
                'VALUES (%s, NULL, NULL, NOW(), NULL, NULL, NULL);',
                [dataset_id])
    db.commit()
    return


def main(arguments=None):
    args = parse_arguments(arguments)
```

```
    conf = args.config
    config = get_conf(conf)
    # Establish DB connection
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    dataset_id = args.metafile.replace('.txt', '')
    metafile = parse_file(args.metafile)
    db_date_requested(db, dataset_id)
    db_insert_metadata(db, dataset_id, metafile, config.path_incoming)
    return



def parse_arguments(arguments):
    parser = argparse.ArgumentParser(description='')
    parser.add_argument('metafile',
                        help='File containing dataset metadata')
    parser.add_argument('config',
                        help='Path to configuration file')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

getmetadata.py

getmetadata.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/other/getmetadata.py>.

```
#!/usr/bin/env python3.4


import requests
import pymysql
import argparse
from configparser import ConfigParser
from collections import namedtuple
import sys
import os
import lega_mirroring.scripts.datasetlogger



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object

    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'api_url': config.get('api', 'api_url'),
            'api_user': config.get('api', 'api_user'),
            'api_pass': config.get('api', 'api_pass'),
            'path_gridftp': config.get('workspaces', 'receiving'),
            'extensions': config.get('func_conf', 'extensions')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object

    :hostname: address of mysql server
    :username: username to log in to mysql server
    :password: password associated with :username: to log in to mysql
server
    :database: database to be worked on
    """
    db = pymysql.connect(host=hostname,
                         user=username,
```

```
                        passwd=password,
                        db=database)
    return db



def request_dataset_metadata(api_url, api_user, api_pass, did):
    """
    This function does an HTTP request to EGA API to get
    dataset metadata and returns it as a stream

    :api_url: address of ega api_url
    :api_user: username to access api
    :api_pass: password for :api_user:
    :did: dataset id
    """
    api_url = api_url.replace('DATASETID', did)
    metadata = requests.get(api_url,
                            auth=(api_user, api_pass),
                            stream=True)
    if not metadata:
        raise Exception('\n\nAPI Error'
                        '\napi_url=' + api_url +
                        '\napi_user=' + api_user +
                        '\napi_pass=' + api_pass +
                        '\ndataset_id=' + did +
                        '\n\n')
    return metadata.json()



def db_insert_metadata(db, metadata, did, path_gridftp, ext):
    """
    This function takes a stream of metadata as input
    and inserts it into database

    :db: database connection object
    :metadata: stream of metadata from EGA API
    :did: dataset id
    :path_gridftp: path to receiving directory
    :ext: crypto-extensions
    """

    cur = db.cursor()
    n_bytes = 0
    n_files = 0

    for i in range(len(metadata)):

        filename = os.path.basename(metadata[i]['fileName'])
        filename = os.path.join(path_gridftp, filename)

        # Removes crypto-extension if it exists
        if filename.endswith(ext):
            filename, extension = os.path.splitext(filename)

        n_files += 1
        n_bytes += int(metadata[i]['fileSize'])
        params_file = [metadata[i]['fileId'], filename,
                       int(metadata[i]['fileSize']), metadata[i]['check-
sum'],
                       'pending']
        params_data = [did, metadata[i]['fileId']]
```

```
        params_log = [did, n_files, n_bytes]

        try:
            cur.execute('INSERT INTO file VALUES '
                        '(%s, %s, %s, %s, %s);', params_file)
            cur.execute('INSERT INTO filedataset VALUES '
                        '(%s, %s);', params_data)
            cur.execute('INSERT INTO dataset_log VALUES '
                        '(%s, %s, %s, NOW(), NULL, NULL, NULL);',
params_log)
        except:
            pass

        i += 1

    db.commit()
    return



def main(arguments=None):
    args = parse_arguments(arguments)
    conf = args.config
    config = get_conf(conf)
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    ext = tuple(config.extensions.split(','))
    metadata = request_dataset_metadata(config.api_url, config.api_user,
                                        config.api_pass, args.dataset)
    db_insert_metadata(db, metadata, args.dataset, config.path_gridftp,
ext)
    return



def parse_arguments(arguments):
    parser = argparse.ArgumentParser(description='This script does an HTTP'
                                     ' request to EGA API to get dataset'
                                     ' metadata')
    parser.add_argument('dataset',
                        help='dataset id, e.g. EGAD00000...')
    parser.add_argument('config',
                        help='path to config.ini')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

datasetlogger.py

datasetlogger.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/scripts/datasetlogger.py>.

```
#!/usr/bin/env python3.4


import requests
import pymysql
import sys
import argparse
import os
import json
from configparser import ConfigParser
from collections import namedtuple



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'path_metadata': config.get('workspaces', 'metadata'),
            'api_url': config.get('api', 'api_url'),
            'api_user': config.get('api', 'api_user'),
            'api_pass': config.get('api', 'api_pass')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object
    :hostname: address of mysql server
    :username: username to log in to mysql server
    :password: password associated with :username: to log in to mysql
server
    :database: database to be worked on
    """
    db = pymysql.connect(host=hostname,
                         user=username,
                         passwd=password,
                         db=database)
    return db
```

```
def read_json(jsonpath):
    '''
    This is for mockup ega api, it reads .json files
    read_api() is the actual api request
    '''
    metadata = 0
    n_files = 0
    n_bytes = 0
    with open(jsonpath, encoding='utf-8') as data:
        metadata = json.loads(data.read())
    for i in range(len(metadata)):
        n_files += 1
        n_bytes += metadata[i]['fileSize']
    return (n_files, n_bytes)



def db_dataset_exists(db, dataset_id):
    exists = False
    cur = db.cursor()
    cur.execute('SELECT dataset_id '
                'FROM dataset_log '
                'WHERE dataset_id=%s',
                [dataset_id])
    result = cur.fetchall()
    if cur.rowcount >= 1:
        exists = True
    return exists



def db_date_requested(db, dataset_id, n):
    cur = db.cursor()
    n_files = n[0]
    n_bytes = n[1]
    params = [dataset_id, n_files, n_bytes]
    cur.execute('INSERT INTO dataset_log '
                'VALUES (%s, %s, %s, NOW(), NULL, NULL, NULL);',
                params)
    db.commit()
    return



def db_date_download_start(db, dataset_id):
    cur = db.cursor()
    cur.execute('UPDATE dataset_log '
                'SET date_download_start=NOW() '
                'WHERE dataset_id=%s;',
                [dataset_id])
    db.commit()
    return



def db_date_download_end(db, dataset_id):
```

```
    cur = db.cursor()
    cur.execute('UPDATE dataset_log '
                'SET date_download_end=NOW() '
                'WHERE dataset_id=%s;',
                [dataset_id])
    db.commit()
    return



def db_date_processing_end(db, dataset_id):
    cur = db.cursor()
    cur.execute('UPDATE dataset_log '
                'SET date_processing_end=NOW() '
                'WHERE dataset_id=%s;',
                [dataset_id])
    db.commit()
    return



def db_date_is_null(db, dataset_id):
    status = False
    cur = db.cursor()
    cur.execute('SELECT date_download_start '
                'FROM dataset_log '
                'WHERE dataset_id=%s',
                [dataset_id])
    result = cur.fetchall()
    if cur.rowcount >= 1:
        for row in result:
            if row[0] is None:
                # if query is empty, date is null
                status = True
    return status



def main(arguments=None):
    """
    This function runs the script
    :arguments: contains parsed command line parameters
    """
    args = parse_arguments(arguments)
    dataset_id = args.dataset_id
    method = args.method
    config = get_conf(args.config)
    # Establish database connection
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    # Check if dataset is already added to log
    if (db_dataset_exists(db, dataset_id)):
        if method == 'date_download_start':
            # If date is NULL on column, update date, else don't
            if db_date_is_null(db, dataset_id):
                db_date_download_start(db, dataset_id)
        elif method == 'date_download_end':
            db_date_download_end(db, dataset_id)
        elif method == 'date_processing_end':
            db_date_processing_end(db, dataset_id)
```

```
        else:
            print('Invalid method: ' + method + ' for dataset: ' + da-
taset_id)
    else:  # If not, create new entry
        if method == 'date_requested_mockup':
            # this is for mockup EGA API
            # aka .json files in /metadata
            # put /path/ and .json to EGAD
            path = args.dataset_id + '.json'
            path = os.path.join(config.path_metadata, path)
            n = read_json(path)  # n_files and n_bytes
            db_date_requested(db, dataset_id, n)
    return



def parse_arguments(arguments):
    """
    This function parses command line inputs and returns them for main()
    :method: parameter that determines the operation of the script
    :path: path to .json
    :config: full path to config.ini (or just config.ini if
                    cwd: lega-mirroring)
    """
    parser = argparse.ArgumentParser(description='Logs dataset processes '
                                     'to database with timestamps.')
    parser.add_argument('method',
                        help='determines which operation date is logged. '
                        '\nPossible values are: '
                        '\ndate_requested(DEPRECATED, now in get-
metadata.py) '
                        '\ndate_requested_mockup '
                        '\ndate_download_start '
                        '\ndate_download_end '
                        '\ndate_processing_end')
    parser.add_argument('dataset_id',
                        help='EGAD0000....')
    parser.add_argument('config',
                        help='path to configuration file.')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

TransferTracking.py

TransferTracking.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/workflows/TransferTracking.py>.

```
import luigi
import lega_mirroring.scripts.monitor



class CheckFilesInDirectory(luigi.Task):
    # Luigi class for WORKFLOW 1


    branches = luigi.Parameter()
    branch = luigi.Parameter()
    config = luigi.Parameter()


    def run(self):
        ''' This function starts the file transfer checking process
        as described in lega_mirroring/scripts/monitor.py '''
        lega_mirroring.scripts.monitor.main([self.branches,
                                             self.branch,
                                             self.config])
        return
```

ProcessMaster.py

ProcessMaster.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/workflows/ProcessMaster.py>.

```
import luigi
import os
import shutil
from configparser import ConfigParser
from collections import namedtuple
import lega_mirroring.scripts.monitor
from lega_mirroring.workflows.TransferProcessing import UpdateFileStatus



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'path_processing': config.get('workspaces', 'processing')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def par(branches, branch, pathr):
    """
    This function reads a directory and generates a list
    of files to be checked by a single parallel process
    :branches:  number of branches to be run
    :branch:  id of branch
    :pathr:  path_receiving from config.ini
    """
    complete_set = []  # to be appended
    selected_set = []  # to be appended
    for root, dirs, files in os.walk(pathr):
        for item in files:
            # form full path
            fullpath = os.path.join(root, item)
            # strip path (to leave subdirs if they exist)
            relpath = fullpath.replace(pathr, '')
            complete_set.append(relpath)
    i = 0
    while i <= len(complete_set):
        index = branch+i*branches
        if index <= len(complete_set):
            selected_set.append(complete_set[index-1])
        i += 1
    return selected_set
```

```
# luigi starts from here



class Launch(luigi.Task):
    # Luigi class for starting TransferProcessing WORKFLOW


    # Remove output folder if it exists
    if os.path.exists('output'):
        shutil.rmtree('output')


    branches = luigi.Parameter()
    branch = luigi.Parameter()
    config = luigi.Parameter()


    def requires(self):
        conf = get_conf(self.config)
        path = conf.path_processing
        selected_set = par(int(self.branches), int(self.branch), path)
        for filename in selected_set:
            filepath = os.path.join(path, filename)
            yield UpdateFileStatus(file=filepath, config=self.config)
```

TransferProcessing.py

TransferProcessing.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/lega_mirroring/workflows/TransferProcessing.py>.

```
import luigi
import os
from configparser import ConfigParser
from collections import namedtuple
import lega_mirroring.scripts.res
import lega_mirroring.scripts.md5
import lega_mirroring.scripts.move
import lega_mirroring.scripts.update



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object
    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'extensions': config.get('func_conf', 'extensions'),
            'path_receiving': config.get('workspaces', 'receiving'),
            'path_processing': config.get('workspaces', 'processing'),
            'path_archive': config.get('workspaces', 'end_storage')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



class DecryptTransferredFile(luigi.Task):
    # WORKFLOW 2 STAGE 1/6


    file = luigi.Parameter()
    config = luigi.Parameter()


    def run(self):
        conf = get_conf(self.config)
        ext = tuple(conf.extensions.split(','))
        if self.file.endswith(ext):  # if file is crypted, decrypt it
            lega_mirroring.scripts.res.main(['decrypt',
                                             self.file,
                                             self.config])
        with self.output().open('w') as fd:
            fd.write(str(self.file))
        return


    def output(self):
```

```
        return luigi.LocalTarget('output/1.txt')



class VerifyIntegrityOfDecryptedFile(luigi.Task):
    # WORKFLOW 2 STAGE 2/6


    file = luigi.Parameter()
    config = luigi.Parameter()


    def requires(self):
        return DecryptTransferredFile(file=self.file, config=self.config)


    def run(self):
        conf = get_conf(self.config)
        ext = tuple(conf.extensions.split(','))
        filename_decr = self.file  # .bam
        # remove crypt extension from filename
        if self.file.endswith(ext):
            filename_decr, extension = os.path.splitext(self.file)
        md5 = lega_mirroring.scripts.md5.main(['check',
                                               filename_decr,
                                               self.config])
        if not md5:
            raise Exception('md5 mismatch')
        with self.output().open('w') as fd:
            fd.write(str(self.file))
        return


    def output(self):
        return luigi.LocalTarget('output/2.txt')



class EncryptVerifiedFile(luigi.Task):
    # WORKFLOW 2 STAGE 3/6


    file = luigi.Parameter()
    config = luigi.Parameter()


    def requires(self):
        return VerifyIntegrityOfDecryptedFile(file=self.file,
                                              config=self.config)


    def run(self):
        conf = get_conf(self.config)
        ext = tuple(conf.extensions.split(','))
        filename_decr = self.file  # .bam
        # remove crypt extension from filename
        if self.file.endswith(ext):
            filename_decr, extension = os.path.splitext(self.file)
        lega_mirroring.scripts.res.main(['encrypt',
                                         filename_decr,
                                         self.config])
        with self.output().open('w') as fd:
```

```
            fd.write(str(self.file))
        return


    def output(self):
        return luigi.LocalTarget('output/3.txt')



class CreateHashForEncryptedFile(luigi.Task):
    # WORKFLOW 2 STAGE 4/6


    file = luigi.Parameter()
    config = luigi.Parameter()


    def requires(self):
        return EncryptVerifiedFile(file=self.file, config=self.config)


    def run(self):
        conf = get_conf(self.config)
        ext = tuple(conf.extensions.split(','))
        # remove crypto-extension
        if self.file.endswith(ext):
            filename, extension = os.path.splitext(self.file)
        lega_mirroring.scripts.md5.main(['hash', filename, self.config])
        with self.output().open('w') as fd:
            fd.write(str(self.file))
        return


    def output(self):
        return luigi.LocalTarget('output/4.txt')



class ArchiveFile(luigi.Task):
    # WORKFLOW 2 STAGE 5/6


    file = luigi.Parameter()
    config = luigi.Parameter()


    def requires(self):
        return CreateHashForEncryptedFile(file=self.file, config=self.con-
fig)


    def run(self):
        conf = get_conf(self.config)
        # Create new directory to end storage location
        # remove /root/path
        relpath = self.file.replace(conf.path_processing, '')
        if os.path.dirname(relpath):
            if not os.path.exists(os.path.join(conf.path_archive,
                                               os.path.dirname(relpath))):
                os.mkdir(os.path.join(conf.path_archive,
                                      os.path.dirname(relpath)))
        ext = tuple(conf.extensions.split(','))
```

```
        if self.file.endswith(ext):
            base, extension = os.path.splitext(self.file)  # remove exten-
sion
            cscfile = base + '.cip.csc'
            cscmd5 = base + '.cip.csc.md5'
        else:  # .bam
            cscfile = self.file + '.cip.csc'
            cscmd5 = self.file + '.cip.csc.md5'
        lega_mirroring.scripts.move.main([cscfile, cscmd5, self.config])
        with self.output().open('w') as fd:
            fd.write(str(cscfile + '\n' + cscmd5))
        return


    def output(self):
        return luigi.LocalTarget('output/5.txt')



class UpdateFileStatus(luigi.Task):
    # WORKFLOW 2 STAGE 6/6


    file = luigi.Parameter()
    config = luigi.Parameter()


    def requires(self):
        return ArchiveFile(file=self.file, config=self.config)


    def run(self):
        conf = get_conf(self.config)
        ext = tuple(conf.extensions.split(','))
        basefile = self.file  # .bam
        if self.file.endswith(ext):
            basefile, extension = os.path.splitext(self.file)
        basefile = basefile.replace(conf.path_processing, '')
        lega_mirroring.scripts.update.main([basefile, self.config])
        with self.output().open('w') as fd:
            fd.write(str(self.file))
        return


    def output(self):
        return luigi.LocalTarget('output/6.txt')
```

CreateTestFiles.py

CreateTestFiles.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/testingscripts/CreateTest-Files.py>.

```
# step 1 create .bam files
# step 2 hash md5 values for files and save them to .json file
# step 3 encrypt .bam files to .bam.cip, then remove .bam files


import time
import hashlib
import os
import argparse
import sys
import lega_mirroring.scripts.res
import lega_mirroring.scripts.md5
import random


'''
    # end-to-end test tool #

    This is a test tool that creates .bam.cip files
    and .json metadata files.

    Note: There is no configuration file, paths are
    hard coded.

    # step 1 create .bam files
    # step 2 hash md5 values for files and save them to .json file
    # step 3 encrypt .bam files to .bam.cip, then remove .bam files
'''



def step1(amount):
    start_time = time.time()
    for i in range(amount):
        f = open('/data/incoming/gridftp-endpoint/file' + str(i) + '.bam',
'w')
        for j in range(100000):  # increase range to create larger files
            f.write(str(i) + 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit.'
            'Aenean gravida ligula id semper maximus. Etiam sagittis, augue
eget accumsan '
            'posuere, tellus lectus ultrices mi, non dignissim nisl risus
vel orci. Curabitur '
            'consequat lorem mauris, eu efficitur felis ultrices bibendum.
Fusce nec ipsum tincidunt, '
            'varius diam in, venenatis odio. Vestibulum massa lectus, cur-
sus vitae orci vitae, '
            'fringilla tincidunt augue. Vivamus vel massa porta, maximus
mauris sit amet, luctus risus. '
```

```
            'Ut nulla nisi, finibus quis sapien id, viverra congue urna.')
            j += 1
        i += 1
    print('Step [2/4]: generate .bam files, runtime: ' + str(time.time()-
start_time)[:6] + 's')
    return


def step2():
    start_time = time.time()
    path = '/data/incoming/gridftp-endpoint/'
    dirlist = os.listdir(path)
    try:
        while len(dirlist) > 0:
            nfiles = random.randint(3, 9)
            if nfiles > len(dirlist):
                nfiles = len(dirlist)
            egad = random.randint(100000,900000)
            egaf = random.randint(100000,900000)
            f = open('/data/incoming/metadata/EGAD0000' + str(egad) +
'.json', 'w')
            f.write('\n[')
            for n in range(nfiles):
                file = dirlist.pop()
                file = os.path.join(path, file)
                filesize = os.path.getsize(file)
                hash_md5 = hashlib.md5()
                filemd5 = 'md5'
                if os.path.exists(file):
                    with open(file, 'rb') as fi:
                        for chunk in iter(lambda: fi.read(4096), b''):
                            hash_md5.update(chunk)
                        filemd5 = hash_md5.hexdigest()
                f.write(
                        '\n    {"fileId":"EGAF0000' + str(egaf) + '",'
                        '\n    "datasetId":"EGAD0000' + str(egad) + '",'
                        '\n    "fileName":"' + file + '",'
                        '\n    "fileSize":' + str(filesize) + ','
                        '\n    "fileMd5":"' + filemd5 + '",'
                        '\n    "fileStatus":"pending"}'
                        )
                if nfiles > 1 and n < (nfiles-1):
                    f.write(',')
                egaf += 1
            f.write('\n]')
            f.close()
    except:
        pass
    print('Step [3/4]: generate .json files (metadata), runtime: ' +
str(time.time()-start_time)[:6] + 's')
    return


def step3():
    start_time = time.time()
    path = '/data/incoming/gridftp-endpoint/'
    for file in os.listdir(path):
        file = os.path.join(path, file)
        lega_mirroring.scripts.res.main(['encrypt', file, 'config.ini'])
        os.rename(file + '.cip.csc', file + '.cip')
        os.remove(file)
    print('Step [4/4]: encrypt .bam files to .bam.cip and remove .bam
files, runtime: ' + str(time.time()-start_time)[:6] + 's')
    return
```

```
def emptyall():
    start_time = time.time()
    path_grid = '/data/incoming/gridftp-endpoint/'
    path_proc = '/data/incoming/processing/'
    path_arch = '/data/incoming/final-archive/'
    path_meta = '/data/incoming/metadata/'
    for file in os.listdir(path_grid):
            file = os.path.join(path_grid, file)
            os.remove(file)
    for file in os.listdir(path_proc):
            file = os.path.join(path_proc, file)
            os.remove(file)
    for file in os.listdir(path_arch):
            file = os.path.join(path_arch, file)
            os.remove(file)
    for file in os.listdir(path_meta):
            file = os.path.join(path_meta, file)
            os.remove(file)
    print('Step [1/4]: clean directories, runtime: ' + str(time.time()-
start_time)[:6] + 's')
    return


def main(arguments=None):
    args = parse_arguments(arguments)
    emptyall()
    step1(args.amount)
    step2()
    step3()
    print(str(args.amount) + ' test files created')
    return



def parse_arguments(arguments):
    parser = argparse.ArgumentParser(description='')
    parser.add_argument('amount', type=int, help='amount of test files')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

TrackItems.py

TrackItems.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/testingscripts/TrackItems.py>.

```
#!/usr/bin/env python3.4


import os
import argparse
import sys


'''
    # lega-mirroring end-to-end test tool #

    This script is used to track moved items.
    python3 TrackItems.py start     # creates a list of files
                                      in /gridftp-endpoint/
    python3 TrackItems.py end       # creates a list of files
                                      in /final-archive/
    python3 TrackItems.py compare   # creates a list of differences
                                      between start and end

    Note: This is a quick test tool with no configuration file,
    so paths are hard coded.
'''


def write_gridftp():
    contents = os.listdir('/data/incoming/gridftp-endpoint/')
    f = open('contents_gridftp.txt', 'w')
    for item in contents:
        f.write('%s\n' % item)
    f.close()
    return




def write_archive():
    contents = os.listdir('/data/incoming/final-archive/')
    f = open('contents_archive.txt', 'w')
    for item in contents:
        item = item.replace('.csc', '')
        f.write('%s\n' % item)
    f.close()
    return



def compare():
    fgrid = open('contents_gridftp.txt', 'r')
    contents_grid = fgrid.readlines()
    fgrid.close()
    farc = open('contents_archive.txt', 'r')
```

```
    contents_arc = farc.readlines()
    farc.close()
    missing_items = set(contents_grid) - set(contents_arc)
    f = open('missing_items.txt', 'w')
    for item in missing_items:
        f.write('%s' % item)
    f.close()
    return



def main(arguments=None):
    args = parse_arguments(arguments)
    if args.cmd == 'start':
        write_gridftp()
    elif args.cmd == 'end':
        write_archive()
    elif args.cmd == 'compare':
        compare()
    else:
        print('invalid command')
    return



def parse_arguments(arguments):
    parser = argparse.ArgumentParser(description='')
    parser.add_argument('cmd', help='must be start, end or compare')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

RequestDatasets.py

RequestDatasets.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/testingscripts/Request-Datasets.py>.

```
#!/usr/bin/env python3.4


import pymysql
import argparse
from configparser import ConfigParser
from collections import namedtuple
import sys
import json
import os
import ntpath
import lega_mirroring.scripts.datasetlogger



def get_conf(path_to_config):
    """
    This function reads configuration variables from an external file
    and returns the configuration variables as a class object

    :path_to_config: full path to config.ini (or just config.ini if
                     cwd: lega-mirroring)
    """
    config = ConfigParser()
    config.read(path_to_config)
    conf = {'host': config.get('database', 'host'),
            'user': config.get('database', 'user'),
            'passwd': config.get('database', 'passwd'),
            'db': config.get('database', 'db'),
            'metadata': config.get('workspaces', 'metadata')}
    conf_named = namedtuple("Config", conf.keys())(*conf.values())
    return conf_named



def db_init(hostname, username, password, database):
    """
    This function initializes database connection and returns a connection
    object that will be used as an executale cursor object

    :hostname: address of mysql server
    :username: username to log in to mysql server
    :password: password associated with :username: to log in to mysql server
    :database: database to be worked on
    """
    db = pymysql.connect(host=hostname,
                         user=username,
```

```
                        passwd=password,
                        db=database)
    return db



def request_dataset_metadata(egad):
    """
    This function opens a .json file and reads the
    values and returns them as a list

   :egad: dataset id
    """
    # read dataset metadata from .json file
    with open(egad, encoding='utf-8') as metadata:
        values = json.loads(metadata.read())
    return values



def db_insert_metadata(db, metadata):
    """
    This function inserts given metadata to database

    :db: database connection object
    :metadata: metadata read from .json file
    """
    cur = db.cursor()
    for i in range(len(metadata)):
        params_file = [metadata[i]['fileId'], metadata[i]['fileName'],
                       int(metadata[i]['fileSize']), metadata[i]['fileMd5'],
                       metadata[i]['fileStatus']]
        params_data = [metadata[i]['datasetId'], metadata[i]['fileId']]
        try:
            cur.execute('INSERT INTO file VALUES '
                        '(%s, %s, %s, %s, %s);', params_file)
            cur.execute('INSERT INTO filedataset VALUES '
                        '(%s, %s);', params_data)
        except:
            pass
        i += 1
    db.commit()
    return



def main(arguments=None):
    args = parse_arguments(arguments)
    conf = args.config
    #egad = args.dataset_id
    config = get_conf(conf)
    # Establish DB connection
    db = db_init(config.host,
                 config.user,
                 config.passwd,
                 config.db)
    # add all datasets and related files to database
    for file in os.listdir(config.metadata):
        file = os.path.join(config.metadata, file)
        db_insert_metadata(db, request_dataset_metadata(file))
```

```
        # strip /path/ and .json to get EGAD
        dataset = ntpath.basename(file)
        dataset_id = dataset.replace('.json', '')
        lega_mirroring.scripts.datasetlogger.main(['date_requested_mockup',
                                                   dataset_id,
                                                   conf])



def parse_arguments(arguments):
    parser = argparse.ArgumentParser(description='This script reads dataset'
                                     ' metadata from json and inserts it to'
                                     ' a database table')
    parser.add_argument('config',
                        help='path to config.ini')
    return parser.parse_args(arguments)



if __name__ == '__main__':
    RETVAL = main()
    sys.exit(RETVAL)
```

FileCount.py

FileCount.py source file is also available on github at <https://github.com/CSCfi/lega-mirroring/blob/master/testingscripts/FileCount.py>.

```
#!/usr/bin/env python3.4


import os


'''
    # simple test tool #

    This script is a simple test tool to quickly check
    file count in working directories.
    Note: There is no configuration file, the paths
    are hard coded.
'''


def fc():
    grid = len(os.listdir('/data/incoming/gridftp-endpoint/'))
    pro = len(os.listdir('/data/incoming/processing/'))
    arc = len(os.listdir('/data/incoming/final-archive/'))
    meta = len(os.listdir('/data/incoming/metadata/'))
    print('gridftp-endpoint: ' + str(grid) + ' files')
    print('processing: ' + str(pro) + ' files')
    print('final-archive: ' + str(arc) + ' files')
    print('metadata: ' + str(meta) + ' files')
    return


fc()
```