



AGTC

genetic tax. consultants

Software Design Document

Christian Buskirk

Peter Bellagh

Chris Blazer

Jorden Kreps

Curtis Rose

Faculty Mentor/Project Sponsor Viacheslav “Slava” Fofanov, PhD

Glossary	2
Introduction	3
Who we are	3
Our Client	3
Big Data	4
Leveraging of High Performance Computing	4
The Problem	5
The Solution: Orchard	5
Implementation Overview	6
Architectural Overview	7
Architecture Diagram	7
Architecture Discussion	8
Key responsibilities and features	8
Main communication mechanisms and information flows	9
Architectural Influence	10
Module and Interface Description	10
Overall UML Diagram	10
Driver Module	10
Setup Module	11
Input Files	11
Input File Formats	12
Luigi Generator	13
Branching	14
Running	15
Implementation Plan	16
Implementation Schedule	16
Conclusion	17

1. Glossary

Configuration File - A file that contains data about a specific user, program, computer, or other file. They are generally read at startup by the operating system and other applications in order to customize the environment for the user.

FASTA - The standard text-based file format used in bioinformatics, where each nucleotide or amino acid is represented using an alphabet of 'A', 'C', 'G', and 'T'.

FASTQ - A merging of the FASTA format with the read quality data given by the sequencing machine, both still using single characters per base.

HPC Cluster - A "High-Performance Computing Cluster"; a set of connected computers that work together such that they can be viewed as a single high performance system or "supercomputer".

Luigi - A Python module that helps build complex pipelines for batch jobs. It handles dependency resolution, workflow management, and visualizations. This is the pipeline management tool that we have selected for this project.

Orchard - A pipeline creation tool with a command-line interface. Assists in creating proper branches and file structures across multiple runs of the same dataset.

Pipeline - A pipeline consists of a chain of processing elements arranged so that the output of each element is the input of the next.

Read Set - Genetic information pulled from a biological sample, produced by High Throughput Sequencing Platform(s). Here this is data in FASTQ format, produced by Illumina brand machines (<http://www.illumina.com/>). This will serve as the primary set of 'queries' for metagenomic analysis that is at the core of this project.

Reference Database - A database that holds a large number of genetically classified organisms. Read sets are queried against these to identify organisms in biological samples. This will serve as the primary 'subject' set for metagenomic analysis.

Scheduling System - Software that assigns processes to resources on HPC clusters, such as time (known as walltime), RAM, and CPU's. The software can terminate processes that use too much of any given resource.

SLURM - An open source HPC cluster management and job scheduling system. SLURM allocates resources to users interested in using a computing cluster, so that many parties can have access to its cores and computing power.

State System - A system/program that can be in one of a finite number of states. The system/program can transition from one state to another. In our case, the system can transition, linearly, from one state to another while saving the output of those states into files.

Taxonomic Classification - Classification of reads (biological sample data) into the species they belong to.

UML - Unified Modeling Language, a general-purpose modeling language used to provide a standard way of visualizing the design of a system.

Walltime - The amount of time allocated to a process by a scheduler in which the process must be completed. If the process does not finish before the walltime is reached, the process will be terminated.

2. Introduction

2.1. Who we are

We, the AGTC Genetic Taxonomic Consultants capstone team, are a team of undergraduate students at Northern Arizona University tasked with creating a pipeline management system to handle the massive amount of data generated and analyzed during the taxonomic classification of metagenomic samples. We are:

- Christian Buskirk - Team Lead
- Peter Bellagh - Administrative Assistant
- Chris Blazer - Chief Communication Officer
- Jordan Kreps - Computing Cluster Specialist
- Curtis Rose - Luigi Specialist

2.2. Our Client

The Fofanov Bioinformatics Lab is part of the School of Informatics, Computing, and Cyber Systems at Northern Arizona University. The lab has a focus on the identification of microbes and other life forms in a given sample through the use of High Throughput Sequencing technologies that allow for the determination of all genomes present in a sample simultaneously instead of checking for each potential pathogen individually. This process generates a tremendous amount of data per sample and the problems inherent in the processing of large amounts of data led to them contacting us through Dr. Viacheslav “Slava” Fofanov and the Northern Arizona University Capstone program in search of a potential solution.

2.3. Big Data

Bioinformatics is the science of collecting and analyzing complex biological data. Recent advances in Genomics, particularly in the area of High Throughput Sequencing, have produced machines capable of producing billions of bases (characters) of genetic code in a matter of days (see Figure 2.1). Thus, even a single small sample can contain genetic information well in excess of a terabyte, after which that data must be queried against reference databases of several hundred gigabytes in order to match the individual genetic strings present in the sample with their matching references for identification, and to correctly classify the sequence. This process can be made manageable mainly through the leveraging of HPC clusters to process the terabytes of data present at any given point in time. Through this analysis, newer fields, such as pathogen tracking, are able to make greater progress than ever before, allowing for a level of tracking specific variants of diseases that was previously not possible.



Figure 2.1: Illumina MiSeq, NextSeq, and HiSeq Sequencing Systems.

2.4. Leveraging of High Performance Computing

A common approach in dealing with the large data sets found in Bioinformatics is to leverage larger HPC clusters to aid in processing the terabytes of data that need to be handled. These computers can have hundreds of individual cores present, with several terabytes of RAM available. However these clusters come with another issue of their own: that of scheduling. Due to the large demand for the power of these HPC clusters to be applied to many different potential research topics, advanced scheduling systems are

utilized to run smaller tasks as space becomes available. Each task is allocated a certain number of cores, amount of memory, and amount of walltime, after which the process will be released from memory and another begins. Any research that is done by our client in this HPC cluster context, therefore, must be able to address the issues caused by dealing with the scheduling program itself as well as any issues created by limitations such as wall time and potential losses that may be caused if violated.

2.5. The Problem

The client has a set of modules that are used to analyze DNA data. The running of these modules can take up to three weeks on a HPC cluster. If the run does not complete in the time allotted, valuable resources are wasted. This system needs to be managed in such a way to save these resources. It needs to be able to pick up where it left off if the run does not complete. It needs to be able to run sets of modules without rerunning all of them if certain parameters are changed. And finally, it needs to be easily usable and maintainable. In its current form, the system is a set of individual modules with no central manager.

2.6. The Solution: Orchard

The solution to this set of problems is to create a pipeline manager that controls the execution, branching, and state saving for the existing system while making it easy to use and easily maintainable. We will be using Luigi, a 3rd party pipeline manager, to control the execution of the modules. We will design our solution, **Orchard**, to take in configuration and link files that contain details about the modules and the current run of the system, and then generate a set of Luigi files accordingly. Orchard will automatically branch when necessary based on changes made to the configuration file. The branching will be handled using an intuitive folder structure, configuring “branched” Luigi files with appropriate paths that ensure it will not execute redundant modules. Luigi intrinsically solves the state saving problem, it does not run tasks if their output file already exists in the working directory. While Luigi is the pipeline manager, Orchard is the Luigi file generator. It will dynamically create Luigi files to manage the pipeline efficiently and effectively.

In short, though we are using the third party branch manager Luigi, Luigi does not meet all the requirements for this project. Our solution, **Orchard**, will meet the following requirements that are not met by Luigi:

- Branch management
- Easy configuration of runs
- Modularity at a pipeline level
- Tracking of previous runs
- Easy to use documentation

3. Implementation Overview

Orchard is a pipeline creation tool that creates a Luigi pipeline that then runs our client's project. Luigi is a pipeline management system developed by Spotify that is used in various applications and controls the flow of a project by handling the inputs, outputs and dependencies between different modules. Our pipeline creation tool will take, as input, a configuration file and a link file that it uses to create the Luigi pipeline. The configuration file defines the inputs and outputs of every module in the project while the link file defines the dependences between the modules. These two files define a Luigi pipeline and our project will interpret these files to create the correct Luigi pipeline. It is important to note that our pipeline creation tool will be able to be used for any project, not just our mentor's project.

Pipeline Link File Example

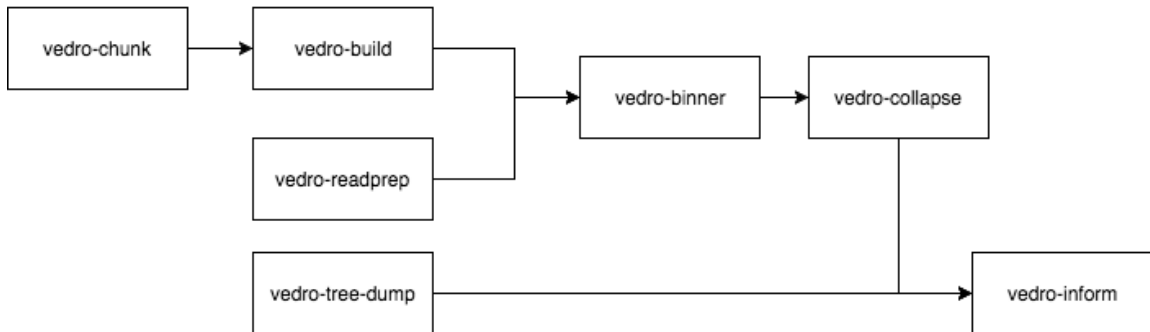


Figure 3.1: Overview of a pipeline linking, showing how each module depends on another

Our client runs his project on the Northern Arizona University's HPC cluster called Monsoon. Monsoon uses a scheduling system called SLURM to handle the allocation of resources to different projects that are being ran at the same time. Some of these resources are memory, time, and number of CPUs. The time limit is known as the "walltime" and it is something we will need to handle. If a module is halfway through writing a file when it reaches the walltime, it will be removed from the HPC cluster and the file is now useless. We will need to implement a way to recognize the difference between a module ending under normal conditions and when one gets stopped because of wall time, out-of-memory issues, or bad configuration files, which result in corrupted outputs. This file will need to be deleted or renamed and moved so that if our pipeline is run at a later date it does not see that the incomplete file exists and determines that it does not need to rerun that module.

4. Architectural Overview

4.1. Architecture Diagram

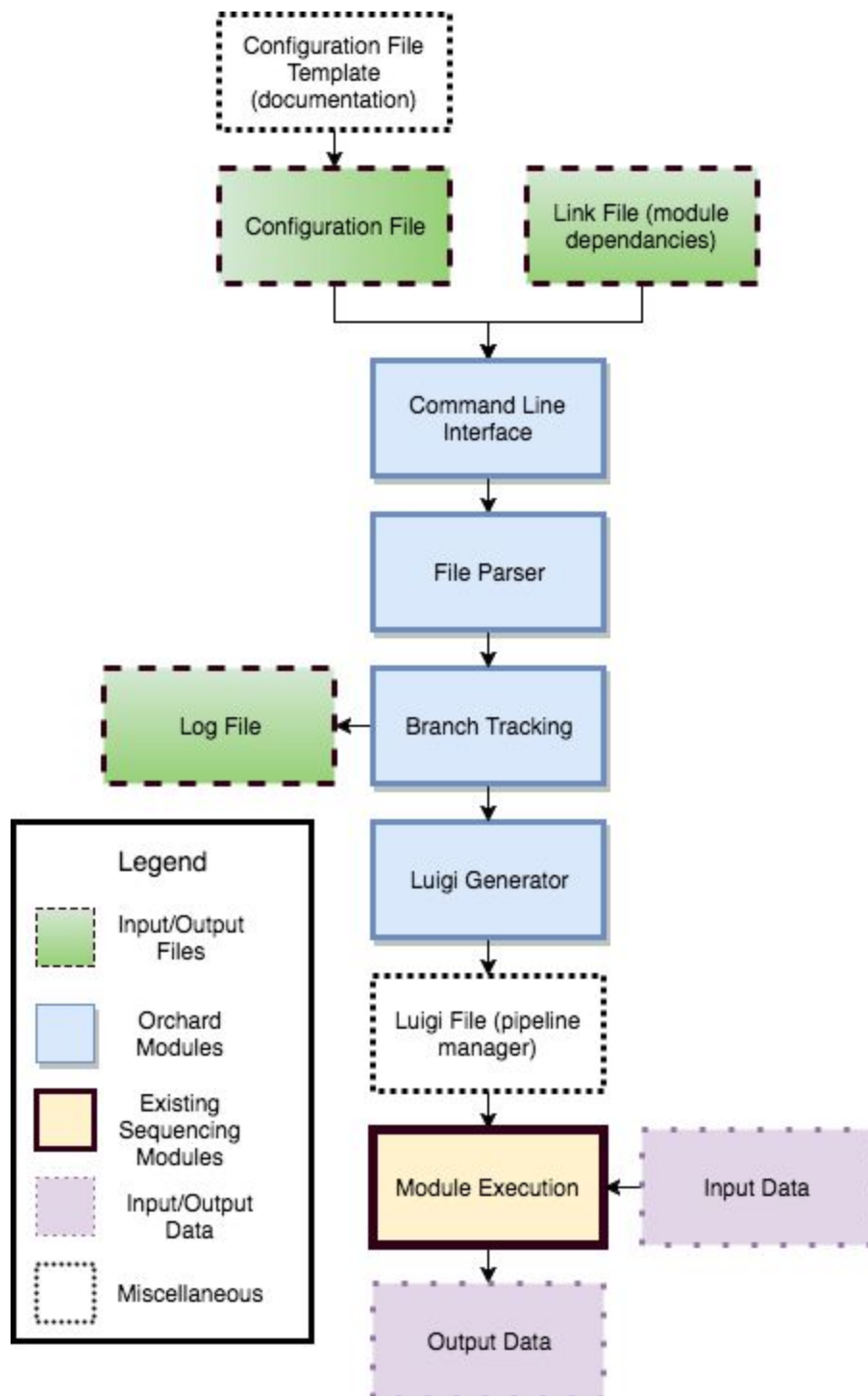


Figure 4.1: Architectural Diagram

4.2. Architecture Discussion

4.2.1. Key responsibilities and features

4.2.1.1. Parse configuration and link files

Our system will parse through configuration and link files that will determine how the pipeline runs. Our system wraps around the already existing modules used by the client. The configuration file will contain information such as source and output paths for each module, as well as sets of options that pertain to each module. The link file will contain all the information regarding module dependencies, and therefore the order in which the modules must be executed.

4.2.1.2. Track branches/manage file structure

Our branch tracking systems main purpose is to dynamically track the state of the pipeline. This will be useful for situations when the user wants to pause a particular run of the pipeline and change one or several arguments in the configuration files. The user will then be able to run this new version of the pipeline and if any of the previously completed modules were created using the same arguments then they will not have to rerun. If the branch tracking system finds a module that used different arguments to run, it will create a branch at this point and start running from there.

To keep track of the different branches, the branch tracking system will create a log file that stores the entire configuration file that resulted in the branch to occur. The log file will also contain useful information including: the time of completion of modules, error messages, and, in general, any information that might be useful to the user.

The main organizational technique that will be utilized is in the directory structure that will be created during a run of the pipeline. There will be a root directory that everything else will be children of. Two completely unique runs of the pipeline will result in two separate directories in the root. If a third run branches off of either of the previous two, a new directory will be added in that previous directory. The newly created directory is where the log file will be stored.

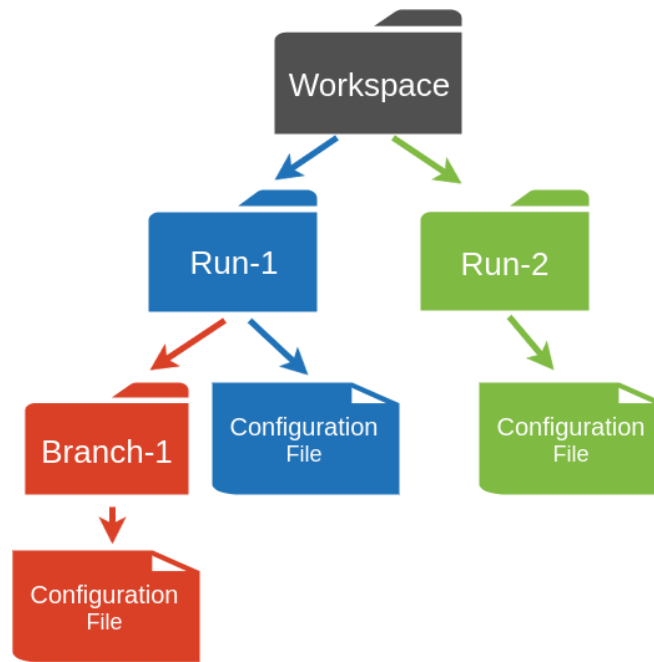


Figure 4.2: File Structure Example

4.2.1.3. Luigi file generation

Our system will create a Luigi file that manages the execution of all modules. The Luigi file will use the information from the parser and branch manager to set up the modules to run in the required order and with the necessary files. Luigi will then handle the execution of all modules.

4.2.2. Main communication mechanisms and information flows

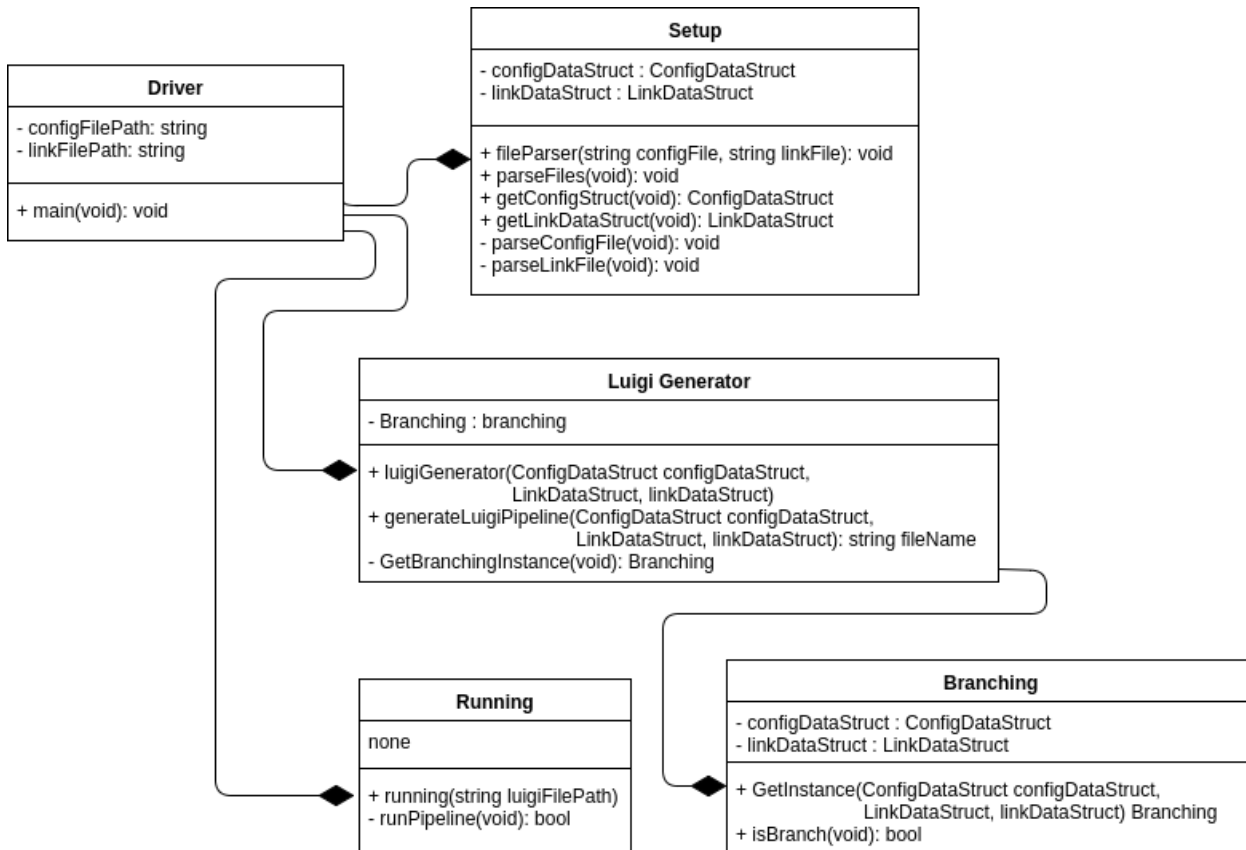
Communication within the system will be based on configuration/link files, directory structure/file paths, and filenames. The configuration and link files contain all the information about how the modules are connected and how they will run, and that informs how the rest of the workflow operates. Luigi uses filenames to determine whether tasks should be run or not. If the output file for a task already exists, Luigi will not run that task. This, in combination with directory structure, will be used to communicate branching in the system. When a new branch is created, a new subdirectory and Luigi file will be generated. The new Luigi file will have a link to the parent directory's files for all modules that do not need to be rerun in the new branch. Then Luigi will look within the subdirectory for the files that pertain to tasks that need to be run, not find them, and execute the required modules in the new branch.

4.2.3. Architectural Influence

Our system is influenced by the pipe and filter software architecture style. The output from each component in the system is used to inform the use of the component that follows it. Starting with the configuration/link files, our system will follow procedures step by step that eventually lead to the creation of a Luigi file that manages the pipeline. Luigi then manages the pipeline in a pipe and filter style as well, with the output from each module being used as input for the next.

5. Module and Interface Description

5.1. Overall UML Diagram



5.2. Driver Module

The primary means of interaction with this software will be through the command line because most of the functionality directed at the HPC cluster that do not have graphical user interfaces. This will be handled through the use of the Python library Click, the “Command Line Interface Creation Kit,” which makes setting up advanced command line tools relatively straightforward. The command line interface will expect two arguments

which are the filepaths of the configuration file and link file. This module will kick off the other modules of the project, acting as the 'main' module.

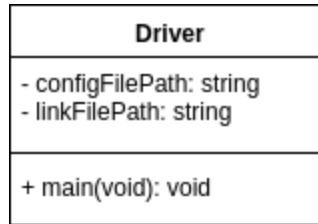


Figure 5.1 UML for the Driver Module

5.3. Setup Module

The Setup module will use the arguments passed into the Driver Module module and parse through them and store them into memory in a custom data structure for each. This information will be used by the Luigi Generator module (Section 5.3) to create the appropriate Luigi file. The input files and their formats are described below.

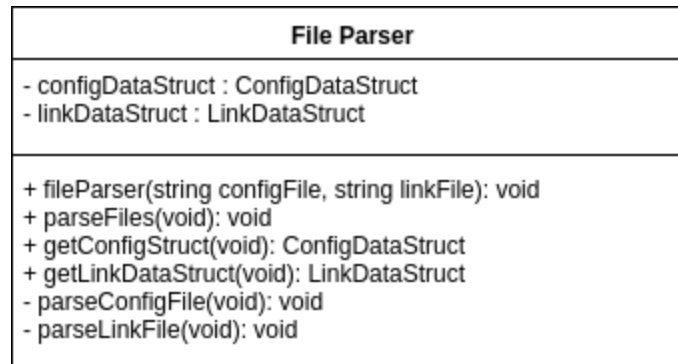


Figure 5.2: UML for the Setup module

5.3.1. Input Files

Orchard will utilize two different input files. The first file is a configuration file that contains the input arguments for each of the modules in the user's pipeline. The second file is a link file that contains the dependency information of the user's pipeline modules. These two files together define a pipelines structure, the arguments necessary to run the modules, and the order in which the modules must be run. Both of these files are created by the user but must follow the format that we define, in detail, below (Section 5.2.2).

The link file will be stored on the cluster, in the user's allotted application support directory, which Click has a built-in method to retrieve the location of, dependent

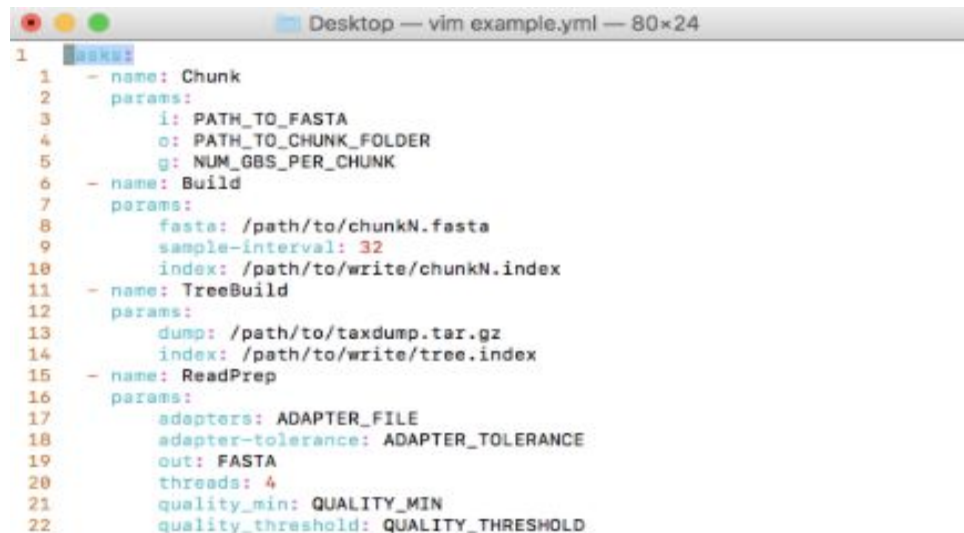
on the platform. From there it will be accessed upon running the build process, which will be discussed later on in this document. The link file will have both the current API of the Bioinformatics pipeline used by Dr. Fofanov, and the relationships between the modules, declaring that pipeline module X will need module Y ran beforehand, and to use those generated resources. This file will drastically reduce any manual configuration required by the end-user in declaring dependencies between their modules.

If the API changes or a dependency is added or dropped there will be methods to retrieve the current configuration, edit it however the user desires, and another command to read in the updated configuration and store it as the new link file to be used on each build.

5.3.2. Input File Formats

5.3.2.1. Configuration File

The configuration file will contain a list of keyword-value pairs that represent the input arguments for the modules and will be organized by module name. The format will be YAML which is a human readable data serialization language. Python has libraries that we can implement that read this file format easily.



```
1  #####
2  - name: Chunk
3    params:
4      i: PATH_TO_FASTA
5      o: PATH_TO_CHUNK_FOLDER
6      g: NUM_GBS_PER_CHUNK
7  - name: Build
8    params:
9      fasta: /path/to/chunkN.fasta
10     sample-interval: 32
11     index: /path/to/write/chunkN.index
12  - name: TreeBuild
13    params:
14     dump: /path/to/taxdump.tar.gz
15     index: /path/to/write/tree.index
16  - name: ReadPrep
17    params:
18     adapters: ADAPTER_FILE
19     adapter-tolerance: ADAPTER_TOLERANCE
20     out: FASTA
21     threads: 4
22     quality_min: QUALITY_MIN
23     quality_threshold: QUALITY_THRESHOLD
```

Figure 5.3: Example configuration file

5.3.2.2. Link File

The link file will also be a YAML format file with keyword-value pairs. These keyword-value pairs will be organized by module and will list the dependency information that Luigi requires to create a pipeline. From this data Orchard will not only know how to generate a proper Luigi file, it will be able to validate a user's configuration file and let them know if something is incorrect.



```
Desktop — vim link.yml — 80x24
1 - vedro-chunk:
  1   params:
  2     ...
  3 vedro-binner:
  4   params:
  5     ...
  6   requires: [vedro-chunk, vedro-readprep]
7 - vedro-build:
  8   params:
  9     ...
 10  requires: [vedro-chunk]
11 - vedro-collapse:
 12  params:
 13    ...
 14  requires: [vedro-binner]
15 - vedro-inform:
 16  params:
 17    ...
 18  requires: [vedro-tree-dump, vedro-collapse]
19 - vedro-readprep:
 20  params:
 21    ...
22 - vedro-tree-build:
:|
```

Figure 5.4: Example link file

5.4. Luigi Generator

After the configuration and link files are read into memory by the Setup module, the Luigi Generator will create the Luigi file. During the generation of this file, the generator will call the Branching module (Section 5.4) as many times as there are modules in the link files. This module determines if a particular module is a branching location or not. Having determined if a particular module is a branching location, a new directory in the proper location that will be used to store the output files for this branch of the pipeline. It will also be necessary to create a new config file that represents this branch and new file locations. This requires that the file paths that are given in the configuration file be altered to point to this new directory as applicable. This is an easy way to keep the output files for different branches separated and organized as best as possible while also preserving Luigi's ability to determine if a file exists or not.

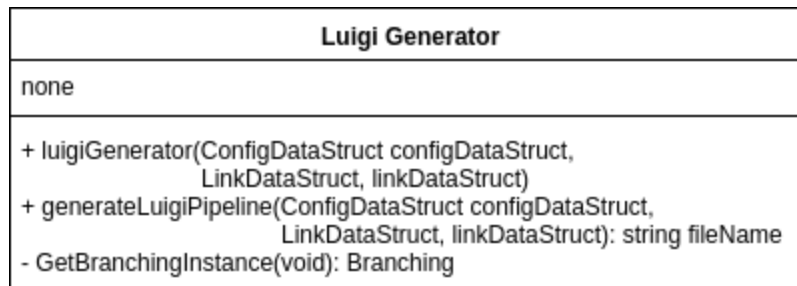


Figure 5.5: UML for the Luigi Generator

```

analysis-workspace — -bash — 122x49
├── binned
│   ├── chunk1.txt
│   ├── chunk2.txt
│   ├── chunk3.txt
│   ├── chunk4.txt
│   └── chunk5.txt
├── branch-chunk-020620171828
│   ├── branch-indices-020620171837
│   │   └── history.yml
│   └── chunks
│       ├── chunk1.fasta
│       ├── chunk10.fasta
│       ├── chunk2.fasta
│       ├── chunk3.fasta
│       ├── chunk4.fasta
│       ├── chunk5.fasta
│       ├── chunk6.fasta
│       ├── chunk7.fasta
│       ├── chunk8.fasta
│       └── chunk9.fasta
│       └── history.yml
├── chunks
│   ├── chunk1.fasta
│   ├── chunk2.fasta
│   ├── chunk3.fasta
│   ├── chunk4.fasta
│   └── chunk5.fasta
├── config.yml
├── history.yml
├── indices
│   ├── chunk1.index
│   ├── chunk2.index
│   ├── chunk3.index
│   ├── chunk4.index
│   ├── chunk5.index
│   └── tree.index
├── informatives.txt
├── prepped
│   └── prepped_reads.fasta
├── reads
│   ├── sequences1.fasta
│   ├── sequences2.fasta
│   ├── sequences3.fasta
│   ├── sequences4.fasta
│   └── sequences5.fasta
└── 8 directories, 37 files
[18:37] analysis-workspace $

```

Figure 5.6: Example file structure with branching folders

5.5. Branching

The Branching module will be used to determine whether or not a specific input module is a branching location. If it is a branching location, this module returns the filepath of the branching location and the Luigi Generator will create a new directory in that location to store all of the files this run of the pipeline generates. This method determines if a specific module in the pipeline is a branching location by looking at all of the existing configuration files. Each configuration file defines its own pipeline structure and argument inputs. It looks for exact copies of modules from beginning to end; the first module that is different is the branching location. If the first module is different then the

entire pipeline will be ran. If all modules are the same then the entire pipeline will not be ran.

There will be some information in the configuration files that might be different than another configuration file but should not result in a branch. This data will be demarcated by a symbol to represent this fact. Examples of data that will not result in a branch are the number of CPU's, the number of threads, and the amount of walltime allocated for use on Monsoon. Any data that is not demarcated in this way can result in a branch.

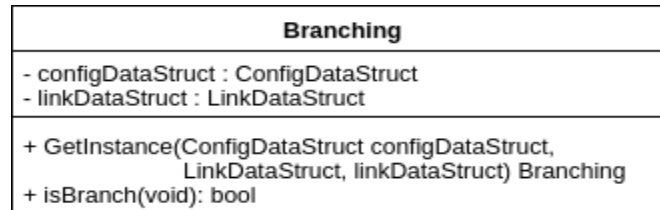


Figure 5.7: UML for the Branching module

5.6. Running

There will be a module that, after the Luigi file is generated, will run that file. This is important because we do not want the user to ever deal with Luigi themselves. With that being said, a Luigi file is being created and will be deleted after the pipeline has finished running. Orchard will take a boolean input that allows the user to specify whether or not the file is deleted at the end or not. The argument defaults to false meaning the file is deleted if the user does not specify otherwise. This can be used for debugging purposes or any other purpose the user finds.

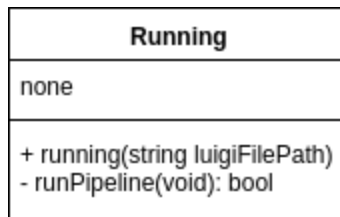


Figure 5.8: UML for the Running module

6. Implementation Plan

6.1. Implementation Schedule

In order to ensure that the project reaches the desired implementation goals, the following schedule, seen in Figure 6.1, was decided upon.

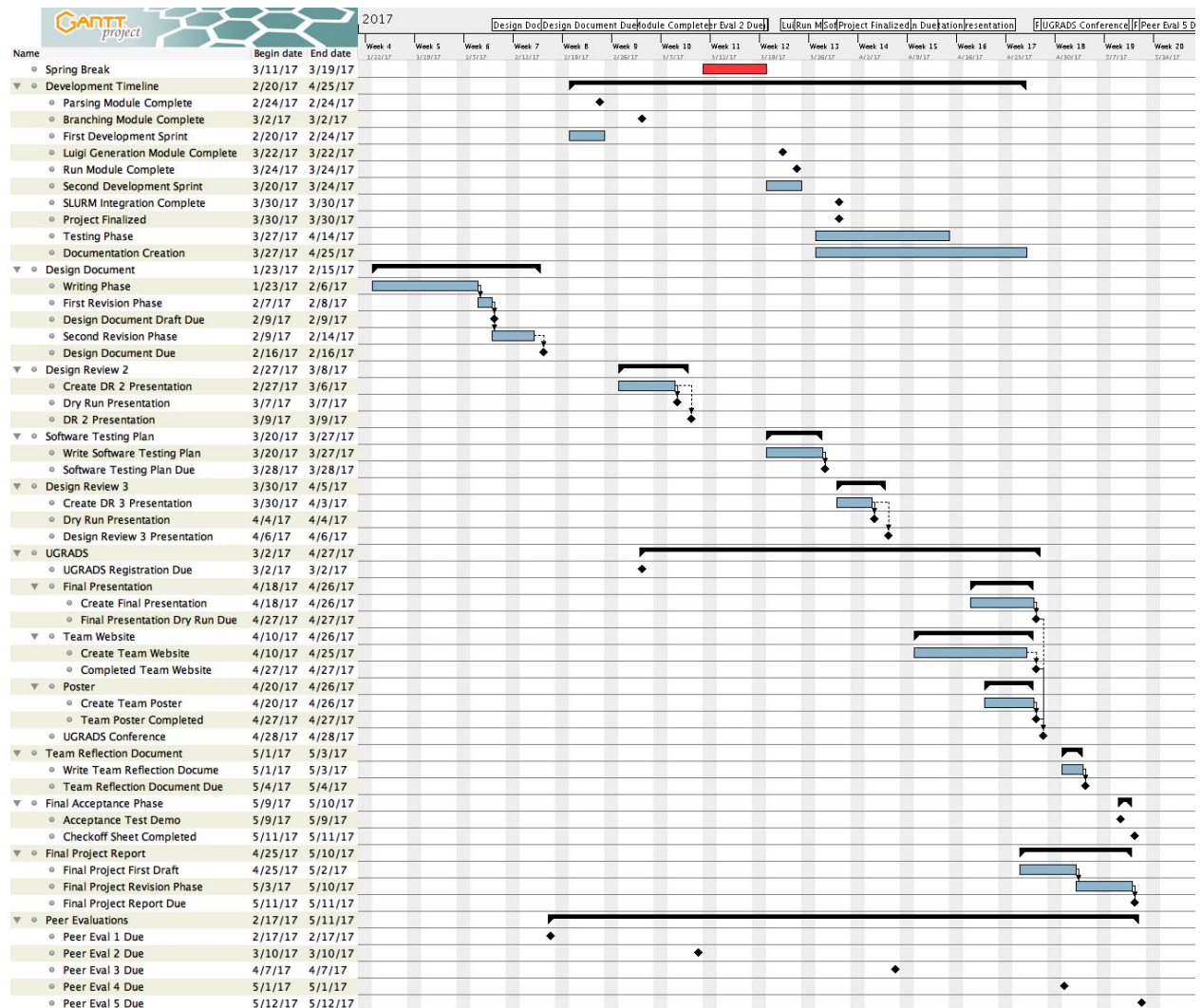


Figure 6.1: Project Schedule Chart

Under the current schedule the majority of our work will be able to be accomplished in two week long sprints, located on the 20th of February and the 20th of March. During the first sprint we hope to be able to accomplish the total completion of the parsing module complete with the command line calls as well as the majority of work required for the branch analysis module to be functional. Over the following two weeks we plan to finish work on the branch analysis module, beginning work on the Luigi generation module prior to breaking for a week during spring break. After spring break the second week-long development sprint will occur, during which time the Luigi generation module

will be completed and the majority of final integration work with the SLURM system can be performed. At this point the system will be relatively finalized, moving into only the testing and bug fixing stage over the last few weeks while the documentation is created and reworked until determined sufficient.

During this period several other milestones will also be occurring, notably the Design Review Presentations 2 and 3 on March 9th and April 6th respectively, as well as the final UGRADS Conference on April 28th. These will be accomplished during lighter periods in the implementation schedule that revolves around the two week-long development sprints.

7. Conclusion

Our client works in the field of bioinformatics and is in need of a pipeline management system that can aid him and his lab while trying to sequence biological samples. This system must be able to perform a multitude of specific tasks in respect to data flow and management to meet the requirements for our final product. These tasks include saving the state of runs, the ability to go back and rerun tasks as branches, the ability to track these branches, and to have a file management system that tracks and saves all of the data generated by the all of these runs. The pipeline management tool that we will be using for the core of our system is called Luigi; a Python module that helps build a complex pipeline of batch jobs. This core then handles the dependency resolution, workflow management, and visualizations for the wrapped pipeline, while our system, Orchard, works as a Python wrapper around it to provide handling of branching and state saving. As the modules currently stand, we as a group are confident that our systems can be implemented successfully, and in their implementation, achieve a solution to our client's problems in sequencing biological samples.