A Cloud-Based Infrastructure for Cancer Genomics

by

Razvan Ioan Panea
Graduate Program in Computational Biology and Bioinformatics
Duke University

Date:_____

Approved:

_____
Sandeep S. Dave, Supervisor

_____
Raluca Gordan

_____
Jen-Tsan Ashley Chi

_____
John Rawls

Dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Graduate Program in Computational Biology and Bioinformatics
in the Graduate School of Duke University

2020

ABSTRACT

A Cloud-Based Infrastructure for Cancer Genomics

by

Razvan Ioan Panea

Graduate Program in Computational Biology and Bioinformatics
Duke University

Date:_____

Approved:

_____

Sandeep S. Dave, Supervisor

_____

Raluca Gordan

_____

Jen-Tsan Ashley Chi

_____

John Rawls

An abstract of a dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Graduate Program in Computational Biology and Bioinformatics
in the Graduate School of Duke University

2020

Abstract

The advent of new genomic approaches, particularly next generation sequencing (NGS) has resulted in explosive growth of biological data. As the size of biological data keep growing at exponential rates, new methods for data management and data processing are becoming essential in bioinformatics and computational biology. Indeed, data analysis has now become the central challenge in genomics.

NGS has provided rich tools for defining genomic alterations that cause cancer. The processing time and computing requirements have now become a serious bottleneck to the characterization and analysis of these genomic alterations. Moreover, as the adoption of NGS continues to increase, the computing power required often exceeds what any single institution can provide, leading to major restraints in the type and number of analyses that can be performed.

Cloud computing represents a potential solution to this problem. On a cloud platform, computing resources can be available on-demand, thus allowing users to implement scalable and highly parallel methods. However, few centralized frameworks exist to allow the average researcher the ability to apply bioinformatics workflows using cloud resources. Moreover, bioinformatics approaches are associated with multiple processing challenges, such as the variability in the methods or data used and the reproducibility requirements of the research analysis.

Here, we present CloudConductor, a software system that is specifically designed to harness the power of cloud computing to perform complex analysis pipelines on large

biological datasets. CloudConductor was designed with five central features in mind: scalability, modularity, parallelism, reproducibility and platform agnosticism.

We demonstrate the processing power afforded by CloudConductor on a real-world genomics problem. Using CloudConductor, we processed and analyzed 101 whole genome tumor-normal paired samples from Burkitt lymphoma subtypes to identify novel genomic alterations. We identified a total of 72 driver genes associated with the disease. Somatic events were identified in both coding and non-coding regions of nearly all driver genes, notably in genes *IGLL5*, *BACH2*, *SIN3A*, and *DNMT1*. We have developed the analysis framework by implementing a graphical user interface, a back-end database system, a data loader and a workflow management system.

In this thesis, we develop the concepts and describe an implementation of automated cloud-based infrastructure to analyze genomics data, creating a fast and efficient analysis resource for genomics researchers.

To those whose love and support have shaped me into who I am:
my parents, Mariana and Ioan Panea,
and my sister, Casandra Maria Panea.

Contents

List of Tables

List of Figures

List of Abbreviations

ABC	Activated B-Cell like
BL	Burkitt Lymphoma
CLI	Command Line Interface
CNV	Copy Number Variations
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DLBCL	Diffuse Large B-Cell Lymphoma
EBV	Epstein-Barr Virus
GCB	Germinal Center B-Cell like
GUI	Graphical User Interface
HIV	Human Immunodeficiency Virus
NGS	Next Generation Sequencing
RAM	Random Access Memory
SSH	Secure Shell
WfMS	Workflow Management System

Acknowledgements

My journey as a PhD student has been paved by numerous experiences and it has been guided by many influential and wonderful people. My time and work in graduate school have been defined by both scientific and personal development. My research work has helped me develop a critical thinking mind, stimulating my interests in biology and computer science, while the people who surrounded me have not only influenced and supported me, but have also changed me into a better person. Thus, there are numerous people to whom I would like to express my gratitude.

First, I would like to thank my mentor and advisor, Sandeep Dave. My work in this thesis would have not been possible without all the opportunities, computing resources and data that he has offered me. Sandeep has been an amazing role model for me since the beginning when I joined his lab. His guidance in both research and life has made my Ph.D. a memorable and fulfilling experience.

Second, I would like to thank all the past and present members of the lab for their scientific, technical, and moral support. I am especially grateful to Anupama Reddy for being a great mentor and for guiding me during my computational work, Cassandra Love and Jennifer Shingleton for explaining and producing all the experimental data, and Lanie Happ for her inspiring scientific thinking. Also, I would like to thank and acknowledge the help received from previous members of the lab, Andrea Moffitt, Florian Wagner, and Jenny Zhang.

I want to specifically thank the development team of CloudConductor and its infrastructure, as without their work my thesis projects would have not been possible. I want to thank Alex Waldrop for his immense contribution to CloudConductor and Tushar Dave for his numerous feature implementations. Also, thank you, Qiu Qin, for creating the website and database components of the infrastructure. Lastly, thank you, Clay Parker, for implementing the managing daemon. Working will all of you has shaped me into a better software engineer.

I express my gratitude to my committee members (Raluca Gordan, John Rawls, and Jen-Tsan Ashley Chi) for their guidance during my Ph.D. studies. Also, I would like to thank my Ph.D. program coordinator, Liz Labriola, and Sandeep Dave's lab assistant, Chrissie Goss, for helping me with all the logistics of my Ph.D.

It has been wonderful to be part of the Computational Biology and Bioinformatics program and community. Thanks to all the CBB students, especially for the help and support from Susan Liu, Debraj Ghose, Rachel Kositsky, and Devang Thakkar. Also, thank you Duke University for providing all the resources and being an outstanding community.

Last but not least, I would like to thank my family for their unending love, support, and guidance. Thank you very much mom and dad (Mariana and Ioan) for never stopping to believe in me and always supporting me. I am indebted for all the efforts and sacrifices you have done to help me get where I am right now, and I cannot thank you enough for the education you have offered me! Thank you, my dear sister Casandra, for

being my angel of protection. Thank you for your enormous support, especially during my Ph.D. studies, and for inspiring me to be as successful as you are. Finally, thank you Dan Ristea for all the help, advice, and encouragement that you are giving me. I appreciate each and every member of my family, and I am really grateful for all the great things you have done for me!

CHAPTER 1. Introduction and Background Material

1.1. Cancer

Cancer is a group of diseases characterized by an uncontrolled proliferation of cells. Cancer is a genetic disease and arises from deregulation of genes involved in normal developmental and repair processes.[1,2]

A better understanding of the underlying genetic events that take place during the cancer development will help us provide methods for a more accurate diagnosis of the disease and establish a more efficient and targeted treatment approach.

1.1.1. Cancer as a genetic disease

Cancer arises from a number of different genetic alterations. The most common types of alterations identified in cancer are single nucleotide mutations, small length insertions and deletion, copy number variations and large-scale structural alterations such as translocations.[1,2]

These mutations can be classified as germline or somatic. Germline mutations are inherited genetic variations that are identified in both the tumor and the normal tissue, as opposed to the somatic mutations that are not inherited and are only found in the diseased tissue. Although there are a few cancers where germline mutations can influence the cancer prognosis and progression (e.g. *BRCA* in breast cancer)[3], somatic mutations represent the vast majority of driver events in cancer. Thus, a better understanding of how these somatic mutations are acquired is necessary.

Somatic mutations can comprise both “driver” or “passenger” events. Driver mutations and genes are those that directly contribute to the malignant state. There are two broad classes of driver genes. Oncogenes (e.g. *MYC*) typically harbor activating genetic events that increase the proliferation rate of the cells. Tumor suppressor genes, on the other hand, typically harbor silencing genetic events and can serve to reduce proliferation or DNA repair functions. These critical events must be distinguished from passenger genetic events which also occur somatically, but do not contribute to the malignant phenotype.

1.1.2. Cancer genetics in diagnosis and treatment

Different cancers arise from a combination of different somatic events in oncogenes and tumor suppressor genes. These genetic events can provide critical clues to potential therapies that are likely to be effective. The patterns of genetic alterations can also be specific for different cancers, potentially enabling their application as diagnostic markers. Until recently, the identification of these genetic alterations was a slow, painstaking process. However, the advent of next generation sequencing has greatly accelerated the identification of genetic drivers in cancers (described below).

A better understanding of the genetic foundation of cancers will enable a contribution to the oncogenesis or the disease evolution will allow us to improve the diagnosis and treatment system.

1.1.3. Lymphomas

Cancer is commonly classified based on the primary site where the disease originated, such as breast cancer, blood cancer and skin cancer, or by the tissue type of origin, such as lymphomas, leukemias and sarcomas.

Lymphomas represent a diverse group of malignancies derived from immune cells that collectively affect nearly half a million cases and 250,000 deaths each year world-wide[4]. Lymphomas affect patients of all ages and occur more frequently in immune-compromised (e.g. HIV) patients. There are nearly 50 different lymphomas[5]. For the purpose of this proposal, we focus on two main lymphomas that arise from immune B cells. The first, diffuse large B cell lymphoma (DLBCL) is the most common form of lymphoma. The second, Burkitt lymphoma (BL), is an important model disease that has been studied extensively because of its associations with *MYC* and the Epstein Barr virus (EBV).

1.2. Next-Generation Sequencing

DNA sequencing, or colloquially named “sequencing”, is the experimental process to identify the order of nucleic acids in the provided DNA molecules.

Sanger sequencing[6] is one of the first developed methods of sequencing. It was originally developed by Frederick Sanger in 1977 and remained the mainstay of DNA sequencing for decades. Sanger sequencing was used to sequence the first human genome (“Human Genome Project”)[7]. The Sanger method uses dideoxynucleotides to identify the

correct order of the nucleotides in the DNA sequence. The method is characterized by high accuracy, but low throughput.

In the past decade, a new generation of sequencing technologies have been developed that enable the high-throughput sequencing of DNA. These methods are called "next-generation sequencing" (NGS) methods. The vast majority of studies use the sequencing by synthesis technology developed by Illumina, which remains the leader in this field with over 90% share of the sequencing market. In our work, we have adopted this technology to better understand the genetic alterations in lymphomas.

1.2.1. Sequencing by synthesis technology

Sequencing by synthesis is the NGS technology developed by Illumina that uses fluorescently labeled nucleotides to reproduce the original sequence[8]. Figure 1 depicts an overview of the chemistry of the sequencing reactions that collectively recapitulate the input DNA sequence.

The output of each sequencing experiment is represented by pairs of short sequences named short "reads". These reads can have lengths up to 150 base pairs, depending on the sequencing instrument used and the settings of the sequencing run. As presented in the Figure 1, these short reads are later used to reconstruct the original DNA sequence, typically by aligning them to a reference genome and identifying any variations.

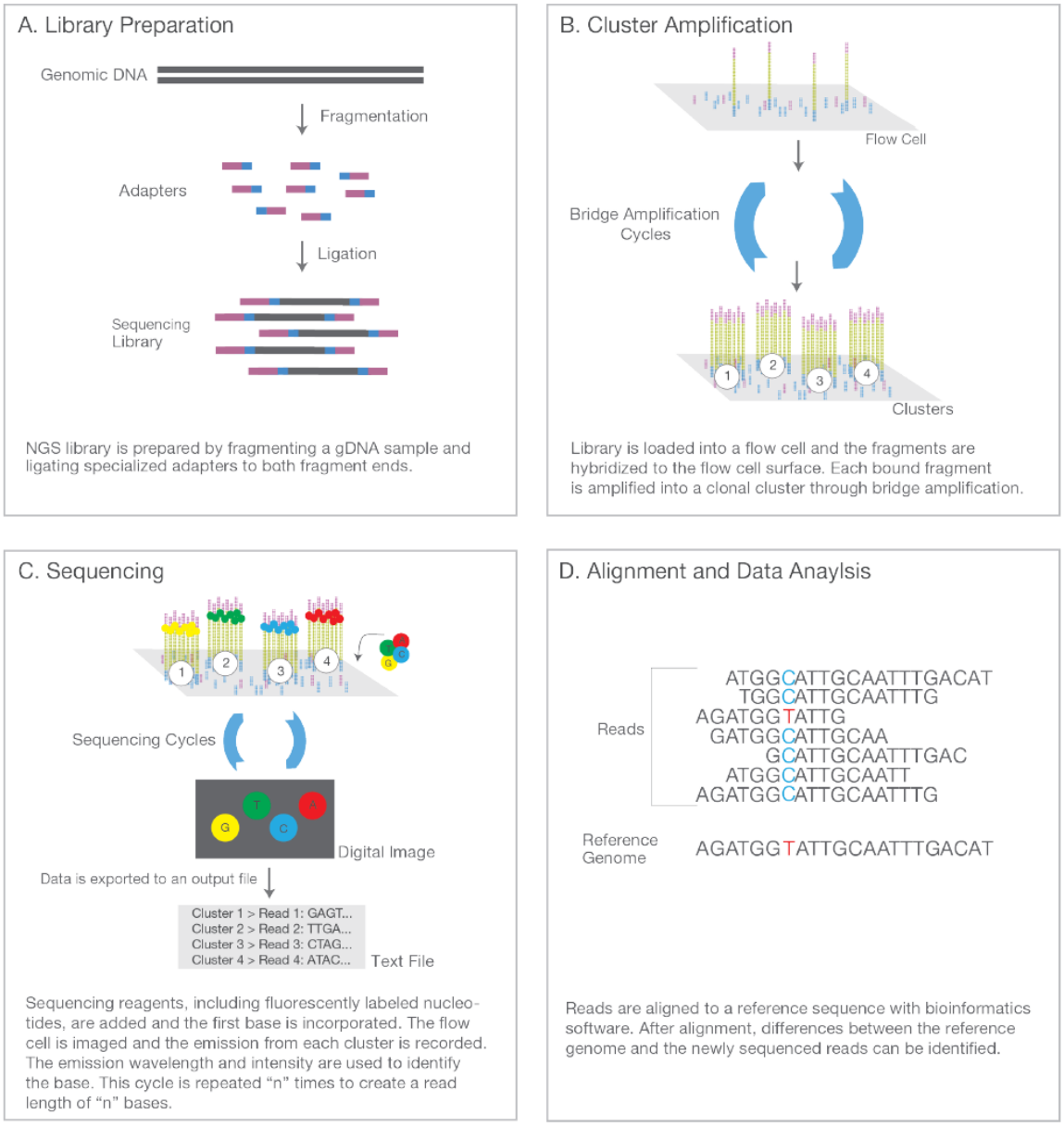


Figure 1. Overview of the chemistry and data analysis in the sequencing by synthesis technology. The DNA fragments are being sequenced using the fluorescently labeled nucleotides.[8]

1.2.2. Scientific applications of NGS

There are a number of scientific applications for short-read sequencing. Here, we describe two widely used approaches relevant to this thesis.

The first approach is DNA sequencing. Whole genome sequencing refers to the process of sequencing unselected DNA from a sample. As the name suggests, it represents sequencing the entire genome. This method is useful not only for comprehensively identifying non-coding variations in the provided genome, but also to discover large scale changes, such as chromosomal alterations, but has the disadvantages of higher cost and poor representation of genomic repeat regions which are challenging for mapping short sequencing reads.

Targeted sequencing is defined by the process of sequencing only certain regions, called the targets, as opposed to sequencing the entire genome. This is typically accomplished by using a set of "baits" that represent the genomic region of interest for hybridization and selective sequencing. Whole exome sequencing is the most popular targeted sequencing method and it represents sequencing only the coding region of the genome (i.e. 1% of the genome). Using exome sequencing, the sequencing reads mostly cover the exons, thus increasing the sequencing coverage on those regions. This method is particularly useful, if the scientific focus is centered on particular parts of the genome (e.g. gene coding regions).

Variant calling is the process of identifying genomic differences in sample of interested when compared to a reference genome. The genomic differences can be both on a small scale, such as single nucleotide variants or small insertions/deletions, or on a large scale, such as translocations. Targeted sequencing of the DNA or whole genome sequencing is commonly used to identify these genomic alterations. The obtained

sequencing reads are aligned to a genomic reference and using specialized bioinformatics tools, differences between the original sequence and the reference genome are reported. Finally, these genomic variants are then filtered for their quality and frequently validated using Sanger sequencing.

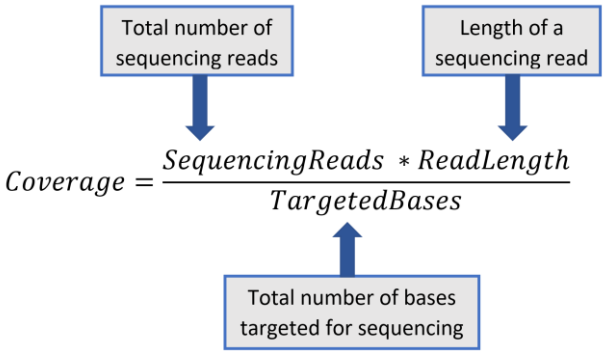
Transcriptome sequencing, also named “RNA sequencing”, is the other main application to sequencing. It represents sequencing only the RNA sequences in the cell, which can then be quantified to define their expression profile. This method typically relies on reverse-transcribing RNA to cDNA and then using the sequencing technology to sequence and quantify these cDNA molecules. The generated sequencing reads are aligned to a reference genome and associated genes, taking into consideration possible splicing events.

There are now dozens of other applications for this technology that are beyond the scope of this thesis. One other application that we have utilized is to use sequencing in conjunction with a genome-scale CRISPR library screen[9] to functionally define the role of different genes in cancers. In this process, each gene from a predefined large set of genes is knocked out using a specific single guide RNA and the Cas9 enzyme. If a specific single guide RNA is found in a higher or lower quantity than the average, then knocking out the targeted gene will result in an increase or, respectively, decrease of the cell proliferation rate. Thus, sequencing the single guide RNAs present in the population and quantifying them will identify the importance of each gene for cell proliferation and provide additional information about the function of genes involved in cancers.

1.2.3. NGS technologies and increasing data demands

Considering the numerous types of questions that can be answered using NGS, it is not surprising that the applications for this technology have exploded. Furthermore, these technologies have become more accessible and affordable, enabling ever more data generation. In this section, we describe how sequencing output is measured and how it affects the asked question.

Sequencing coverage represent the metric for identifying how many base pairs have been sequenced to replicate the original sequence and has the following formula:



For instance, sequencing at 100x coverage implies that if all sequencing reads were distributed uniformly on the sequenced region of interest, then each base pair from the original sequence was sequenced 100 times. The total sequencing coverage can be increased either by reducing the targeted region or by simply generating more sequencing reads. Coverage calculations are necessary to identify how many reads should be sequenced in order to cover the region of interest at a specific depth.

After alignment, sequencing reads are not distributed uniformly on the reference genome, as certain genomic regions may be preferentially selected during the library preparation. For instance, there are numerous mono-, di- or tri-nucleotide repetitive sequences in the human genome, called repetitive regions, which are not only difficult to align to genome, but also require numerous sequencing reads to represent.

Allele frequency refers to the proportion of DNA molecules that represent a variant at a particular genomic locus. In cancer, certain disease-causing alleles may be hard to detect or quantify in the sequencing data because of their low frequency. A low allele frequency can be arisen from numerous factors such as allele heterogeneity, tumor purity and clonality. In order to identify allele with low frequency in the sample, sequencing at a higher depth (i.e. increasing the sequencing coverage) is necessary. For example, for a variant with an allele frequency of 0.01, a minimum depth of 100 reads would be needed to identify the variant.

Increasing the overall sequencing coverage will result in more sequencing reads generated, thus increasing the output size of the data. Given the ever-increasing application of NGS, the data needs for these experiments are also increasing, posing significant challenges in computing infrastructure needs and costs.

1.3. Cloud Computing

As biological data increases in size[10], new adaptable and scalable methods of processing and storage are needed. Even as institutions invest significant resources for

computing servers and infrastructure, these resources frequently fall short of the rate at which sequencing data are being produced.

Considering the increasing application of NGS approaches in cancer, data analysis has become increasingly challenging. First, the size of the sequencing output can become increasingly large, especially when analyzing a large cohort of samples. For instance, a recent large-scale cancer study of whole genomes has generated over 700 terabytes of sequencing data from over 2,500 patients[11]. Furthermore, this sequencing data represents only the input of a complex analysis pipeline that will produce additional intermediate and final data. Thus, an organized method of storing the data is required.

Second, the performance of the bioinformatics tools is directly influenced by the size of the input data. Before analyzing the data to answer the research questions, the input data needs to be processed. In genomics, the processing pipeline requires numerous compute-intensive steps such as alignment, quality checking, variant calling or transcript quantification. Fortunately, most of the bioinformatics tools responsible for these steps are parallelizable, thus their performance can be improved by providing more computational resources. To further improve the performance in some of these processing steps, the input data can be divided into smaller chunks so that they can be processed independently, and their output will be merged into one final output. Finally, in order to process and analyze efficiently a large cohort of samples, a large amount of computing resources is necessary.

1.3.1. The advent of cloud computing in cancer genomics

Cloud computing refers to decentralized computing systems that can be used on-demand for processing. This approach represents a scalable solution for handling the needs imposed by the explosion in NGS applications in biology. Cloud platforms can provide the means for both processing and storing genomics data. Cloud platforms also provide tools for crucial aspects such as security, availability and reproducibility of the analysis. In this section, I will describe the application of cloud for processing genomics data.

1.3.2. Processing and storing genomics data on cloud platforms

There are numerous services provided by cloud platforms, many of which are highly specialized for different applications. In genomics, the two main services that are commonly utilized are “compute” and “storage”.

The compute service represents the actual processing center of the cloud platform and can be used to create “instances” (i.e. the unit element of the compute service). An instance represents a server that is accessible for the user to perform any computing tasks named “processes”.

In bioinformatics, the most used instance types are Linux-based servers. Each instance is mainly characterized by three important computing components. The first component is the total count of central processing units (CPUs). In computing, CPU performs the actual computation on the data. For parallelized computational tasks, a higher number of CPUs would improve performance. The second component of an

instance is the amount of random-access memory (RAM) that can be accessed by the CPUs. This serves as transient, but readily accessible resource. Tasks that require rapid access to a large amount of data usually benefit from instances with large RAM size. The third and final component of an instance is the local storage or disk. The disk is the storage element of the instance that is persistent and has a much slower reading and writing speed comparing to RAM. In general, the disk is used to store the input, output and any intermediate files generated by the computational tasks.

Tasks can vary significantly in their computing resource requirements. Some tasks may require more CPUs, while others require more RAM to be efficient. Thus, it is important to understand the computational analysis and its processing needs before identifying a suitable instance type. Unlike institutional resources, a great advantage of cloud platforms is that requesting computing resources is very simple and the newly created instances can usually be accessed almost immediately.

As the name suggests, the storage service provides the storage-related functionalities. Although cloud platforms theoretically provide unlimited storage, it is important to organize, plan and manage genomic data storage. Besides the total allotted size, there are additional aspects that need to be considered when storing genomic data and its metadata. The level of security with which the data is encrypted and stored is important, especially when clinical data are being stored, as there are numerous regulations, including HIPAA, that need to be respected when storing sensitive information. Additionally, the availability of the data is important when considering how

easy and fast will the data be accessed by the compute services. Finally, data replication is a requirement to ensure that no data loss will ever occur in the future. In general, cloud platforms provide a number of options to manage these tasks, but they require informed choices by the users before they can focus on their actual computational task.

1.4. Workflow Management Systems

A workflow is an automation in which a linear or non-linear sequence of processes are connected by a flow of data communication. Each process produces an output which is provided, fully or partially, as input to the following downstream processes. In computing, a workflow is frequently represented as a directed acyclic graph (DAG), while in bioinformatics it is referred to as “pipeline”.

A workflow engine is a software that interprets an already defined workflow and executes its tasks, while ensuring the data sharing between the tasks. A workflow management system (WfMS) is a framework that generates a workflow, manages its execution and ensures a running environment for the workflow engine.

1.4.1. Workflow management systems in bioinformatics

There are numerous WfMSs available in bioinformatics, such as Toil[12], Nextflow[13] or Butler[14]. In bioinformatics, WfMSs are sometimes referenced as bioinformatics frameworks[15]. Table 1 presents an example set of the currently popular WfMSs and some of their features.

Table 1. List of popular workflow management systems[15].

WfMS name	Syntax	Paradigm	Interaction
Toil	Explicit	Class	CLI
Nextflow	Implicit	Convention	CLI
Apache Airflow (Butler)	Explicit	Configuration	CLI
Snakemake	Implicit	Convention	CLI
Queue (GATK)	Explicit	Class	CLI

Leipzip et al. classifies bioinformatics frameworks using syntax, design paradigm and method of interaction. Syntax can be implicit or explicit, depending on how the workflow tasks are being defined. Implicit syntax frameworks can generate workflows depending on specific rules applied on elements such as filenames, while explicit frameworks require the user to completely specify the order of the tasks and their configuration.

The design paradigm can be conventional which is based on already defined languages such as Make or Python. The design paradigm can also be configuration-based, which is represented by one or multiple explicit configuration files, or class-based that encapsulates most of the configurations into a set of classes. Users can interact with WfMSs either through a command line interface or a graphical interface. The graphical interface, also named workbench, can be a web-based or local application.

There are three critical components that are essential for the functionality of a WfMS. The first component is the scheduler, which is responsible for the processing and execution of the workflow. The scheduler ensures that the provided workflow is a valid graph and its tasks are running in the correct order. The second component is the resource manager, which is responsible for collecting and registering the data produced by each workflow task. It also establishes the data communication between the tasks. The final component is the processing platform which represents the environment where the tasks are actually executed.

1.4.2. A multitude of processing environments for WfMSs

The processing platform is an important component of a WfMS. Workflows can be processed on an individual computer (e.g. Linux or Windows server), on a computing cluster, such as Oracle Grid Engine or SLURM, or on a cloud platform, such as Google Cloud or Amazon Web Services.

There are different approaches to using computational resources that utilize premade groups of computing servers (“clusters”) or more dynamic groups of computing resources that requires more careful administration. These approaches have significant trade-offs in processing cost and the appropriate utilization of computing resources.

Considering that many biological datasets are already available on the cloud and researchers are increasingly adopting the cloud for processing their generated data, numerous modern WfMSs use popular cloud platforms, such as Amazon Web Services or Microsoft Azure, as the main processing environment. However, many institutions and

organizations have their own private cloud or other computing infrastructure which can operate very differently than the well-known public cloud platforms. Although there are bioinformatics frameworks that support more than one cloud platform provider, none of them provide a platform-agnostic solution.

A software solution is platform-agnostic if it can run all its processes on any platform. As it is not feasible to implement any available platform, a platform-agnostic solution would provide an abstract interface that will allows the user to implement any processing platform. Apache libcloud (https://libcloud.apache.org/) is an application programming interface that aims to provide a common language between over 50 cloud-based providers. Although there are potentially important benefits can be obtained from libcloud in bioinformatics, it cannot be used as a direct processing environment as it is not an actual WfMS.

A platform-agnostic solution would ensure that any new computing platform can be considered.

1.5. Cloud-Based Automation

Processing large biological datasets on the cloud is complex. Typical bioinformatics pipelines encompass many steps from securely uploading the input sequencing data to the cloud platform to running different genomics analyses to visualizing the analysis output. Each of these steps include many different substeps and computational processes that need to be carefully considered.

An automated infrastructure is indispensable to enable the feasibility of such analyses. Automation in our case refers to a bioinformatics infrastructure in which the manual human intervention is minimized for the processing and analysis of genomics data. In this section, I will present five important components that are part of an automated infrastructure.

First, the graphical user interface (GUI) is the central communication component necessary for the users to interact with the elements of the infrastructure. The GUI can be built as a dedicated software (i.e. application), using programming languages such as Java or C#, or as a website using web frameworks and platforms like Django or Node.JS. This provides secure interaction with the user and shields the users from aspects of the infrastructure that should not be accessible.

Second, the underlying database is a crucial component for storing biological data, metadata and infrastructure records. It is important to note that the database is not the central storage component of the infrastructure, as biological data files, especially sequencing files, can become quite large. Thus, these files are not stored directly in the database. The database simply references their location on another storage system, such as cloud storage. There are numerous database systems available, such as MySQL or MongoDB. Although relational database management systems have been widely used in bioinformatics, studies show that non-relational databases system can provide a performance improvement[16]. In the end, selecting a database system is mainly dependent on the type of data stored.

Third, the workflow engine represents the central processing element of the infrastructure. This component represents the diverse bioinformatics pipelines that can be executed in different ways. As discussed above, there are numerous WfMSs available in bioinformatics. The selection of the appropriate framework is influenced by factors such as the input data size, the processing platform and the type of analysis pipelines used. Additionally, it is important to understand the design of the selected workflow engine as it can directly affect the other components of the infrastructure.

The fourth component of an automated infrastructure is the data loader, which is responsible for interpreting the metadata present in the database and translating it into the format used by the workflow engine. The loader makes the logical connection between the location of the input data to be processed, the type of bioinformatics analysis to be performed and the configuration of the processing platform.

The fifth and final automation element is the managing daemon, which executes and connects all the infrastructure components. The daemon is a process that runs continuously in the background and performs different tasks. In an automated infrastructure, the managing daemon connects to the database and identify any new workflows generated by the data loader, so that they are provided to the workflow engine. Additionally, the managing daemon continuously reports the status of the running workflows and their generated output to the database. Although the managing daemon performs the main part of the automation, the other components represent the critical foundation of the infrastructure.

In this thesis, I will describe an implemented cloud-based infrastructure in bioinformatics, specifically for cancer genomics, using the described computing components. I will describe the automated infrastructure and CloudConductor (https://github.com/labdave/CloudConductor), a cloud-based bioinformatics workflow management system that we developed. I will continue by showing a real-world application of the analysis and characterization of Burkitt lymphoma subtypes[17] that was made possible using the implemented framework.

CHAPTER 2. CloudConductor: A Cloud-Based Bioinformatics Workflow Management System

2.1. Introduction

CloudConductor is a cloud-based workflow management system that we developed to generate and execute bioinformatics pipelines. This framework is designed to organize and create pipelines. The pipelines are specified as four configuration files that define the pipeline graph, the resources used, the platform information and the input data. The framework interprets the provided configuration files and runs a pipeline generated on-the-fly.

CloudConductor was developed to address three central challenges of computing in bioinformatics. The first challenge is the diversity of methods in bioinformatics analyses. This diversity is reflected in the myriad of different analysis pipelines that are in active use, as well as the different computing resources needed for their successful execution. Thus, any system we implement has to be flexible enough to allow the user to generate any bioinformatics pipeline. Additionally, various computing resources are used by different institutions and research groups, making impossible to set a default processing environment. CloudConductor was designed to allow the researchers to execute their pipelines in any cloud environment.

Secondly, scalability is important in genomics as the input sample sizes have been continuously increasing over time. Existing computing resources such as local and institutional servers and clusters are necessarily limited and have generally not kept up

with the rate at with genomics data have been expanding. Cloud computing platforms such as Google Cloud or Amazon Web Services represent a solution to this problem. However, connecting existing bioinformatics solutions to these cloud-based services remains a major challenge. CloudConductor was specifically designed to leverage the advantages of cloud computing platforms.

Finally, reproducibility remains a serious challenge in data analysis. Thus, preserving the entire version of all the components in an analysis pipeline is crucial. By using "container" technology (Docker), CloudConductor preserves the tools versioning and allows the researchers to run their pipelines anywhere in a reproducible fashion.

By offering solutions to these challenges, we have designed CloudConductor as a comprehensive workflow management system. The system works in three stages. The first stage is represented by the researcher defining the analysis workflow from a set of predefined or user-defined bioinformatics tools. In the next stage, CloudConductor generates a container-based environment and executes these tools in the correct order by allocating resources on a cloud computing platform. Finally, in the last stage, when the processing is complete, the analysis statistics and output will be reported to the user.

In this chapter, I will present the central components of CloudConductor and how these are configured in order to run a pipeline. I will continue by presenting the features of CloudConductor, followed by implementing and running an example pipeline. Finally, I will provide the performance statistics from running the example pipeline. The

documentation (https://cloudconductor.readthedocs.io/en/latest/) and program (https://github.com/labdave/CloudConductor) are available online.

2.2. Component 1: the pipeline

The first component of CloudConductor is pipeline creation. In bioinformatics, a pipeline is defined as a sequence of bioinformatics tools that transform and analyze the input data. A pipeline can be represented as a directed acyclic graph of nodes, in which each node is a bioinformatics tool. In CloudConductor the pipeline is provided as an input through a configuration file named pipeline graph.

2.2.1. Modules and their implementation

A module is a bioinformatics tool that can perform multiple functions, with each function being defined in CloudConductor as a separate submodule. Some bioinformatics tools have only one function, thus their modules have only one submodule. However, most commonly used genomics tools (e.g. Samtools*)* have multiple submodules. Thus, before implementing a tool in CloudConductor, it is important to understand what its functionalities are.

From the programming perspective, a module is represented by an actual Python module, while a submodule is defined as a Python class in the module. Thus, in order to create a module, the user needs to implement a Python class that describes the functionality of the bioinformatics tool and place it in the correct Python module that is associated with the tool.

Figure 2 depicts a schematic Python class, similar to an interface, of a submodule in CloudConductor. When defining a submodule, it is important to first identify its type as each submodule class is inherited from an abstract class defining a type.

Submodules comprise three types, mainly depending on how the input and output data are used by the implement tool. The first and most common type, named simply "tools", is represented by bioinformatics tools that perform a plain transformation of a set of input files and generate a new set of output files, without splitting or merging them. The second type of submodule is "splitters", which represent tools that split one input data entity into multiple fragments of data of the same type. Finally, the third type of submodule is "mergers", which are tools that merge chunks of data of the same type, into one output data entity. Maintaining a distinction between these submodule types is important as they are a vital component for the algorithm that generates pipelines.

Further, submodules are defined by three main elements: the input keys ("arguments"), the output keys and the command. Each input and output key represent a file or value the submodule receives as input or returns as output. For example, for an alignment submodule, one input key is the reference genome, while one output key is the generated alignment file. After the input and output keys are defined, the user can use them to generate the actual command of the submodule. As illustrated in the Figure 2, input keys, output keys and the command need to be defined respectively in their methods "define_input", "define_output" and "define_command".

```python
from Modules import Module


class Submodule(Module):

    def __init__(self, module_id, is_docker=False):
        """
        Initialize the new Submodule class.

        Args:
            module_id (string) - the unique ID generated by CloudConductor for
                                 this object
            is_docker (boolean) - the current submodule should return a docker
                                  specific command
        """
        super(Submodule, self).__init__(module_id, is_docker)

        # Define list of output_keys the command will generate data for
        self.output_keys = ["output_key1", "output_key2", "output_key3"]

    def define_input(self):
        """
        Define the input of the submodule
        """
        pass

    def define_output(self):
        """
        Define the output of the submodule
        """
        pass

    def define_command(self):
        """
        Generate the actual command
        """
        pass

```

Figure 2. Python template for a submodule class. The submodule class is represented by the three defining methods.

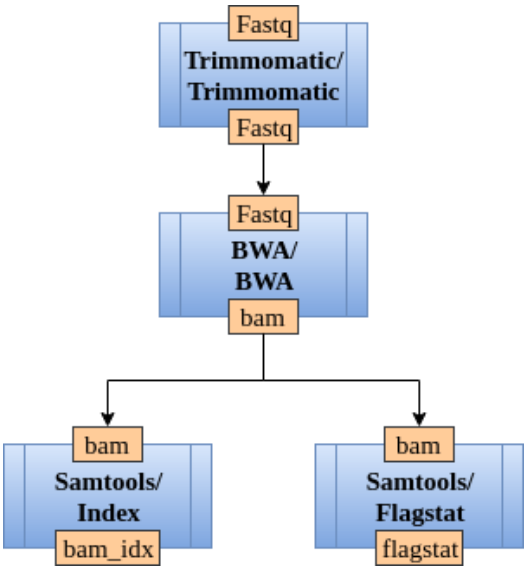


Figure 3. Simple DNA alignment pipeline. The sequencing reads are trimmed and then aligned to the reference genome using BWA. The resulted alignment file is being summarized by Samtools flagstat.

Figure 3 shows a simple example of a pipeline in which the submodule "Trimmomatic" returns "fastq" as an output key which is later received as an input key by the downstream submodule "BWA". Consequently, in order to generate a pipeline, the user must ensure all input keys of a module can receive their data from either the parent submodules or the input sample and resources data.

In order to implement a new submodule, the user needs to understand the input and output requirements of the bioinformatics tool to be integrated. Next, the input and output requirements need to be specified in the "define_input" and "define_output" methods, respectively. The tool command is then generated and returned by the user in the "define_command" method using the input argument values obtained using the "get_argument" method and the output values returned by the "get_output" method.

Figure 5 presents an example of an implemented submodule for obtaining information about alignments using “Samtools”. Four arguments have been specified as input keys for the “Flagstat” submodule. The input key “samtools” represents a resource and is the software that will be retrieved from the resource kit, while the input key “bam” is the data that will be analyzed. The input keys “nr_cpus” and “mem” are necessary for specifying the computing resources necessary for processing. Samtools flagstat returns only one output file that contains general alignment information; thus, the submodule defines only one output key named “flagstat”.

2.2.2. Dynamic pipeline generation using the scheduler

CloudConductor generates and executes the pipelines dynamically. Before executing the pipeline, CloudConductor reads the input configuration file and interprets the defined modules and submodules that are part of the pipeline. After connecting all modules into an initial graph, CloudConductor uses a scheduler to execute the pipeline based on the generated graph.

A scheduler is a program, which is part of the workflow engine, that is responsible for launching, provisioning and monitoring the workflow tasks. The scheduler was implemented to adjust the pipeline graph depending on the type of the submodules that are being executed, specifically if they are splitters or mergers.

Figure 4 presents how CloudConductor generates the pipelines during their runtime. When during execution, the scheduler completes the task of a splitter submodule, then all downstream tasks are being split into multiple branches until the next

merger submodule is present. Thus, the user can split an entire pipeline graph by implementing a splitter and a merger submodule in the correct location of a pipeline.

When using the splitter and merger functionality it is important to ensure the correct order of splitting and merging. The splitting process is executed dynamically as the number of downstream branches and it is dependent on the input data provided to the splitter module. For example, a module that splits the sequencing data in smaller pieces could generate more splits if the input sequencing data is large.

In CloudConductor, the pipeline is managed and generated in a Python class named “Graph”. In this class, the directed acyclic graph of the pipeline is structured using adjacency lists. The adjacency lists can change during pipeline runtime as explain above.

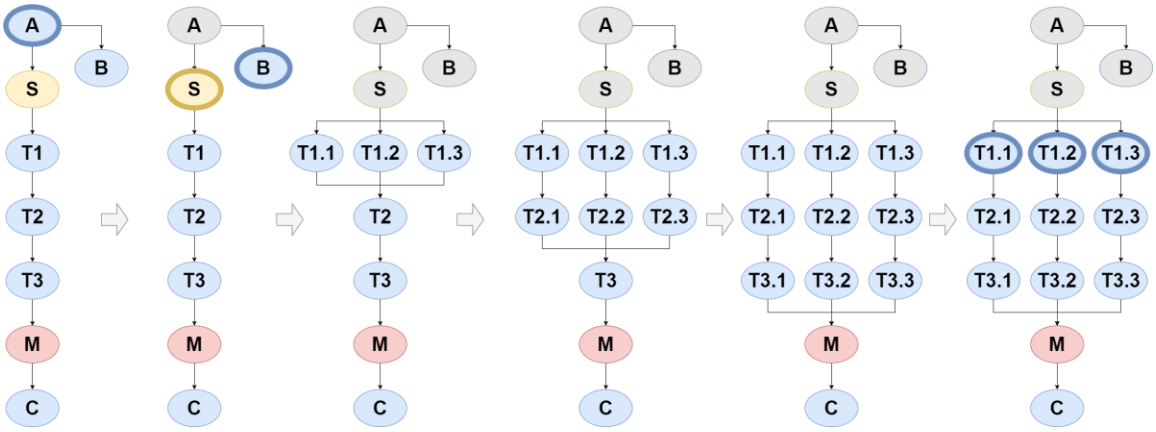


Figure 4. On-the-fly generation of an example scatter-gather pipeline. Nodes “S” and “M” represent the splitter and the merger submodules, respectively. Nodes that have solid border are actively running processes, while nodes that have gray fill color are completed processes.

```python
from Modules import Module


class Flagstat(Module):
    def __init__(self, module_id, is_docker = False):
        super(Flagstat, self).__init__(module_id, is_docker)

        # Define the output keys generated by the submodule
        self.output_keys = ["flagstat"]

    def define_input(self):

        # Define the necessary resources
        self.add_argument("samtools",   is_required=True, is_resource=True)

        # Define the input arguments
        self.add_argument("bam",        is_required=True)

        # Define the computational resources
        self.add_argument("nr_cpus",    is_required=True, default_value=2)
        self.add_argument("mem",        is_required=True, default_value=5)

    def define_output(self):

        # Generate a unique file name and path for the output
        flagstat = self.generate_unique_file_name(".flagstat.out")

        # Define the output
        self.add_output("flagstat", flagstat, is_path=True)

    def define_command(self):

        # Obtain argument values for generating the command
        bam         = self.get_argument("bam")
        samtools    = self.get_argument("samtools")

        # Obtain output path
        flagstat    = self.get_output("flagstat")

        # Generating samtools flagstat command
        cmd = "{0} flagstat {1} > {2}".format(samtools, bam, flagstat)

        return cmd

```

Figure 5. Submodule implementation in CloudConductor for samtools flagstat. The methods “define_input” and “define_output” specifies the input and, respectively, output keys of the submodule. The method “define_command” generates the actual command that will be run by CloudConductor.

2.3. Component 2: the resources

The second component in CloudConductor is the resource management. The resources are defined by a configuration file named “resource kit”. The resource kit outlines the location of the bioinformatics tools and required input resources to run the provided pipeline.

All resources from the resource kit can be divided in two broad groups. The first group is represented by the bioinformatics resources that are available through the containerized system, Docker. In general, this group is composed of all the bioinformatics tools that are being used in the pipeline. It is strongly recommended that the user provide all the bioinformatics tools through Docker to ensure the reproducibility of the pipeline. The second group is defined by resources available on an external storage system such as a cloud storage service. This group usually consists of input files, such as reference genomes or genomic regions of interest, that are necessary for the bioinformatics tools to function.

Each resource is defined by a resource type and a path. The resource type represents a key that should be similar to an input key of a bioinformatics tool defined in a module. The path can be either one external path to an executable (or a file) or it can be an entire directory referenced as a containing directory. There are numerous examples of tools for which just the program is sufficient to launch and complete the application. However, most bioinformatics tools that require an environment of libraries to function. For these tools the entire directory of the essential files needs to be provided. Thus, by

using the containing directory feature, the user can provide also an entire directory of a bioinformatics tool to ensure its functionality.

In CloudConductor, the resource manager is named the “DataStore”. It represents a class that manages the paths and configurations of all the input resources and the output files generated by all the submodules during the workflow execution. Furthermore, the DataStore ensures that all the submodules will be able to obtain their input files, by prioritizing the data from different sources.

2.4. Component 3: the input data

The third component of CloudConductor is the input data, which is represented by the inputs to process. They represent the most variable element between pipelines. In genomics, the input data can be categorized as either sequencing data or analysis data. Sequencing data represent the first raw input type that an analysis pipeline in genomics can start from. After processing the sequencing data, the downstream output data that are generated represent analysis data. Analysis data can be an alignment file, a file containing genomic variants or simply a text file containing quality metrics. The analysis data can also be used as input for further downstream analysis pipelines. It is important to note the difference between the two types of data as their metadata is completely different, i.e. library information is required when specifying sequencing data, while source analysis is required for analysis data. In CloudConductor, the input data has been generalized to receive different types of data as well. Thus, a proteomics researcher can replace sequencing data with mass spectrometry proteomics data.

The input data for a pipeline is specified by a configuration file named “sample sheet” which defines a set of samples for analysis that can be either sequencing or analysis data. As the input data for a pipeline can easily grow into hundreds of samples, it is important to keep it in a structure that allows fast access. CloudConductor processes and manages the input data in a class named “SampleSet”.

2.5. Component 4: the execution platform

The fourth component of CloudConductor is the execution platform. This component is defined using the platform configuration file. There are three elements that need to be considered when configuring and setting up the execution platform.

First, users must select the platform on which they plan to perform the analysis. CloudConductor has been implemented and tested on two popular cloud platforms, Google Cloud Platform and Amazon Web Services. If a different cloud platform is used for processing, the user will need to implement its functionality in CloudConductor. There are three abstract classes, CloudPlatform, CloudInstance and StorageHelper, that enable the users to use their cloud platform of choice.

Second, after the user has set their preferred platform and implemented it in CloudConductor, authentication to the platform must be established. The user must ensure that CloudConductor has the permission to read and write data to the preferred storage system and to create, configure and destroy instances on the compute system of the implemented cloud platform. It is important to have all the above permissions available to enable CloudConductor to function properly.

Finally, an important step when using CloudConductor is understanding resource limitations on the cloud platform. CloudConductor can run thousands of instances and allocate petabytes of storage, but the cloud provider might not be able to provide that scale (especially to new users). Also, the total processing cost will be higher when using multiple resources, even as the results will be generated much faster. It is user's responsibility to select the allocation range of the computing resources as it strongly depends on the prioritization between time and total computing cost.

2.6. Features

CloudConductor has seven main features. In this section, I will present CloudConductor's features and explain how they were implemented.

2.6.1. Parallel

CloudConductor is characterized by a multi-level parallelism. The framework parallelizes the computational tasks at multiple levels to dramatically reduce the processing time.

First, CloudConductor can leverage the parallelism offered by all the bioinformatics tools as it can request instances with large number of CPUs. Next, as shown in Figure 4, CloudConductor can parallelize a process in a "scatter-gather" manner, using the splitter and merger submodules. Another layer of parallelism can be achieved by running multiple bioinformatics tools or "scatter-gather" branches in parallel. Finally, the last level of parallelism is at the sample-level at which the user can

provide multiple samples in the input data and process the same pipeline on each of them in parallel.

Using all these levels of parallelism, CloudConductor can leverage the resources provided by the execution platform to process a pipeline in a rapid manner.

2.6.2. Modular and extensible

CloudConductor was developed to have all its components modularized, thus separating the logic between them. Both the future developers and users can benefit from this feature.

From the developer's perspective, it is easy to change components or add new extensions. For example, all the logic necessary to process a module from a pipeline graph or to transfer data from the resource kit can be found only in the execution platform component. Thus, if any component needs a computing resource, or data transfer, they only need to reference methods already implemented in the platform component.

Furthermore, from the user's perspective, the implementation of a new bioinformatics tool or a new execution platform, can be as easy as the extension of the available abstract classes and implementation of the necessary methods. For example, if the user wants to integrate a new submodule, they only need to create a new Python class as an extension from the abstract classes, Module, Splitter or Merger, depending on the submodule type.

2.6.3. Scalable and elastic

An important feature of CloudConductor is the scalability and elasticity of the framework. This feature is vital considering that in bioinformatics the input data size can vary drastically, depending, for instance, on the input sequencing type or on the type of genomic data used for analysis.

CloudConductor can scale up both horizontally and vertically depending on the implementation of the submodules used in the workflow. CloudConductor can allocate multiple instances to process the data in parallel. By scaling horizontally, it can add additional instances of the same type, while by scaling vertically, it can increase the specification of the used instances to increase the performance. During our test of the scalability feature, CloudConductor was able to scale horizontally to 500 instances of the same type and vertically to instances of 96 CPUs and over 600 GB random access memory. These values are not a limitation of the computing resources that CloudConductor can utilize, but only an illustration of the scalability of the system.

In CloudConductor, the elasticity feature was implemented by ensuring that only the necessary resources are requested. For example, when requesting a disk for an instance, CloudConductor computes the total storage necessary for the tool to successfully run on the specific instance.

Thus, with these features, CloudConductor can not only scale up and down depending on the input data size but can also ensure that only the necessary resources are being used during the scaling process.

2.6.4. Robust and fault tolerant

The design behind CloudConductor's code makes it robust and fault tolerant. A fault-tolerant tool is able to adapt and correct itself when an error appears. CloudConductor achieves robustness and fault tolerance in two ways.

First, the framework performs multiple pre-launch validation checks, to ensure that no infrastructure-related error appears during the pipeline runtime. Before running any command, CloudConductor checks whether all the paths specified in the resource kit or sample sheet do exist. Additionally, each submodule is defined by a fixed set of input and output keys. CloudConductor also checks if the provided pipeline graph is feasible by ensuring that every input key of every submodule can be set during runtime. Thus, using these validation steps, the framework becomes more robust and ensures that an analysis pipeline will not stop because of an error that could have been prevented in the first place.

Second, fault tolerance is accomplished by retrying multiple times to run a command after it has failed. As the processing takes place on a different computing environment than the local environment, there are many situations where a submodule can fail because of infrastructure issues such as a temporary drop in network connection. Thus, by rerunning the failed commands returned by the submodules, CloudConductor ensures that failures are not related to temporary infrastructure issues that can be prevented by a simple retrial. Additionally, checks on the received error messages are performed to identify the type of action that is performed.

2.6.5. Cost efficient

As cost is an important factor when processing large datasets, CloudConductor was implemented to minimize the necessary cost of computation. Besides the elasticity feature, CloudConductor can also leverage the discounted offerings from the cloud platforms.

Popular cloud platforms, such as Google Cloud and Amazon Web Services, provide instances with special characteristics that make them cheaper. For example, Google Cloud is offering a type of instance called “preemptible” that is 20% of the regular price. The difference between preemptible and regular instances is that regular instances are available for unlimited time after their request, while preemptible instances have a maximum lifespan of 24 hours and can be terminated any time after a 90 seconds-long notice. Similarly, Amazon Web Services offer spot instances which are cheaper, but can be interrupted on a short notice.

CloudConductor enables both preemptible and spot instances, and carefully checks the status of the instances on the cloud platform. In case a command fails, we would analyze if the failure is associated with a termination event. If a termination event was identified, we would reset the instance and retry running the command. Although using preemptible and spot instances reduce the cost of the analysis, they also increase the total runtime and adds a degree of variability in the process. Thus, because of this feature, the user can make choices that tradeoff cost and reliability.

Further, the elasticity feature of CloudConductor ensures that only the necessary resources are requested. Additionally, CloudConductor does not provision a cluster before running a pipeline and only creates instances whenever a command needs to be run. Consequently, the user only pays for the resources that are actually necessary for processing.

2.6.6. Portable and reproducible

In research, reproducibility is a critical feature when performing data analysis. To ensure that the pipeline is as deterministic as the pipeline tools allow, CloudConductor uses the container technology. Docker is a container technology that allow the user to containerize all dependencies of a software and to fix their versions. CloudConductor is designed to receive bioinformatics tools through Docker repositories, thus allowing the users to confirm the correct version of the tools used.

Another useful feature provided by Docker is portability. Considering that all tools can be delivered using Docker and that all input data and execution can be placed on a cloud platform, CloudConductor can also be portable. Thus, to run CloudConductor, the users only need to install the framework and have a reliable internet connection.

2.6.7. Platform-agnostic

A significant and powerful feature of CloudConductor is that it is platform-agnostic. Using the provided interfaces, users can implement their preferred computing platforms and storage systems.

To process every module of a pipeline, CloudConductor follows the same cycle. After initializing (i.e. authorizing) the cloud platform, an instance is requested. The instance will have an external IP address and an SSH (secure shell) server installed, configured and running on it. Using plain SSH connections, CloudConductor can send commands to the SSH server available at the provided external IP address. When the commands are finished, CloudConductor will request the platform to destroy the server. When all modules are complete, CloudConductor will clean-up the platform of any intermediate data generated.

Furthermore, there are two levels of processing granularity in implementing CloudConductor to be platform agnostic. The first level of granularity is at the pipeline level in which CloudConductor would read the data from any storage source, process the entire pipeline on one specified cloud provider and write the final output data on any storage source. The second level of granularity is at the module level where CloudConductor would read the data from any storage source, process modules on multiple specified cloud providers (thus transferring data between providers during a pipeline processing) and writing the final output on any storage system. CloudConductor has currently been implemented using the first level of granularity.

To make CloudConductor platform agnostic, we implemented three abstract classes. The first abstract class is “CloudPlatform” and it stores information about authenticating CloudConductor and the new platform and enables the creation of a new instance on the platform. This method is implemented to use the second abstract class.

The second abstract class is “CloudInstance” and it stores all the information regarding each instance on the new cloud platform. In order to use the platform-agnostic functionality, the user must extend these two abstract classes. Thus, an object from the CloudPlatform extended class will generate a set of objects from the CloudInstance extended class.

Further, the third abstract class, named “StorageHelper”, abstracts the cloud storage services. The user will need to extend this abstract class as well if they want to use a special storage system. Currently, CloudConductor automatically identifies which extended StorageHelper class to use based on the protocol of the paths. For example, CloudConductor will use Google’s or Amazon’s StorageHelper implementation if the provided paths have as protocols “gs” or “s3”, respectively.

To extend any of these abstract classes, the user can use the default API system of the cloud provider or a general API solution such as Apache libcloud (https://libcloud.apache.org/). Libcloud is an API provided by Apache that allows the user to perform basic operations on multiple cloud providers. Currently, libcloud supports more than 50 cloud providers including popular platform such as Google Cloud Platform, Amazon Web Services and Microsoft Azure. Furthermore, libcloud is in continuous development, thus new features and newer platforms could be implemented. In CloudConductor, we have used libcloud to create the implementation of the compute services for Google Cloud Platform and Amazon Web Services and can be easily extended to the other cloud providers.

Finally, the platform abstraction has been implemented in a fashion that is not specific to a fully implemented cloud platform. Both normal Linux servers and High-Performance Computing clusters are accessible through SSH allowing them to function essentially as instances on a cloud platform. When implementing these resources as execution platforms, there is no need to actually allocate and deallocate resources, but only provide access to them.

2.7. Example implementation and results

To show an example of a CloudConductor run and to benchmark its performance on real data in a cloud computing environment, we analyzed transcriptome (RNA) sequencing data from 775 tumor biopsies obtained from a recent study[18]. On average there are 24 million read pairs per sample.

Using our framework, we implemented a transcriptome sequencing pipeline that identifies and quantifies transcripts using STAR (as “star_bam” node) and RSEM (as “rsem” node), respectively. We used Picard to identify and mark PCR and optical duplicates. Figure 6 presents the pipeline used. We used Google Cloud with preemptible instances as the main execution platform.

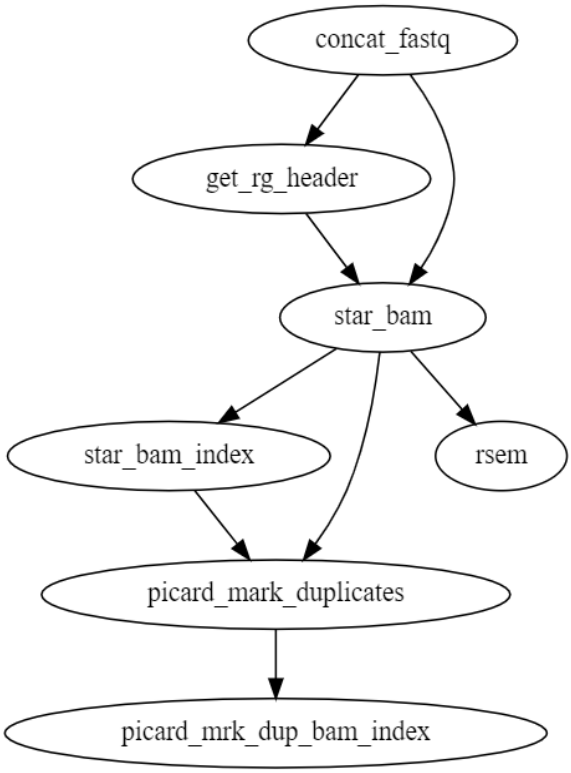


Figure 6. Analysis pipeline used to process 775 transcriptome sequencing samples.

2.7.1. Configuration files

CloudConductor requires four configuration files to successfully run a pipeline. In this section I will present and explain the configuration files used to run the example transcriptome sequencing pipeline.

The first configuration file is the pipeline graph that is used to specify the bioinformatics tools used and their order. Figure 7 shows the pipeline graph configuration file used. We have currently implemented over 50 widely used bioinformatics tools to analyze NGS data, both DNA and RNA, such as BWA, STAR, CellRanger and Diamond. Of these, only 4 are necessary to analyze transcriptome data: STAR is used for read

alignment, RSEM is used for quantification, Samtools Index is used for indexing an alignment file and Picard MarkDuplicates is used to identify and mark sequencing reads that are PCR duplicates.

```
[concat_fastq]
module = Utils
submodule = ConcatFastq

[get_rg_header]
module = Utils
submodule = GetReadGroup
input_from = concat_fastq

[star_bam]
module = Star
docker_image = star
input_from = concat_fastq, get_rg_header
final_output = transcriptome_mapped_bam, raw_read_counts, final_log
    [[args]]
    ref = star_genome_dir

[star_bam_index]
module = Samtools
submodule = Index
docker_image = samtools
input_from = star_bam

[picard_mark_duplicates]
module = Picard
submodule = MarkDuplicates
docker_image = picard
input_from = star_bam, star_bam_index
final_output = bam

[picard_mrk_dup_bam_index]
module = Samtools
submodule = Index
docker_image = samtools
input_from = picard_mark_duplicates
final_output = bam_idx

[rsem]
module = RSEM
docker_image = rsem
input_from = star_bam
final_output = isoforms_results, genes_results

```

Figure 7. Configuration file of the pipeline graph used for transcriptome analysis. Each node of the graph is represented by a section. A node is connected to its parent using the “input_from” field.

```
[Docker]
    [[picard]]
        image = biocontainers/picard-tools:v2.8.1dfsg-1-deb_cv1
        [[[picard]]]
            resource_type = picard
            path = PicardCommandLine
    [[star]]
        image = quay.io/biocontainers/star:2.6.0c--1
        [[[star]]]
            resource_type = star
            path = STAR
    [[samtools]]
        image = quay.io/biocontainers/samtools:1.9--h8ee4bcc_1
        [[[samtools]]]
            resource_type = samtools
            path = samtools
    [[rsem]]
        image = quay.io/biocontainers/rsem:1.3.0--boost1.64_3
        [[[rsem]]]
            resource_type = rsem
            path = rsem-calculate-expression
[Path]
    [[star_genome_dir]]
        resource_type = ref
        path = DATA_PATH/ref/human/hg38/gencode/v29/star
    [[ref]]
        resource_type = ref
        path = DATA_PATH/ref/human/hg38/gencode/v29/GRCh38.primary_assembly.genome.fa*
    [[ref_idx]]
        resource_type = ref_idx
        path = DATA_PATH/ref/human/hg38/gencode/v29/GRCh38.primary_assembly.genome.fa.fai
    [[gtf]]
        resource_type = gtf
        path = DATA_PATH/ref/human/hg38/gencode/v29/annotation/gencode.v29.primary_assembly.annotation.gtf
    [[rsem_ref]]
        resource_type = rsem_ref
        path = DATA_PATH/ref/human/hg38/gencode/v29/rsem/GRCh38.gencode_v29.primary_assembly*
```

Figure 8. Configuration file of the resource kit used for transcriptome analysis. In this figure, “DATA_PATH” is a placeholder for the path to the external storage where all the reference files are present.

The second configuration file is the resource kit which is used to specify all the input resources required for processing the transcriptome pipeline. Figure 8 shows the configuration file of the resource kit. The resource type is represented in the resource kit by the mandatory keyword “resource_type”. It has the same value as the input key of a pipeline module that uses the specific resource. For example, submodule “Index” from module “Samtools” has one input key, named “samtools” that defines the required

resource type. Consequently, the resource of type “samtools” will be connected to the input key “samtools” from submodule “Index”.

The executable path is represented in the resource kit by the mandatory keyword “path”. Its value should be the executable path that CloudConductor will execute.

Sample sheet is the third configuration file used to specify the sample data on which the analysis will be performed. Figure 9 shows the sample sheet used for one single sample analysis, however a sample sheet can contain an indefinite amount of samples.

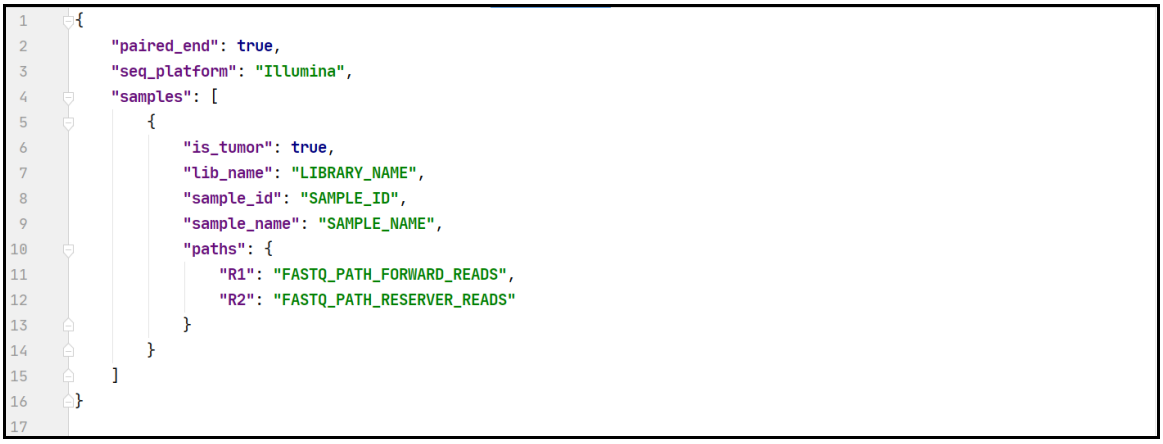


Figure 9. Configuration file of the sample sheet used for transcriptome analysis. In this configuration file, all sample data is anonymized by descriptive placeholders.

Finally, the platform configuration file is used to define the execution platform. Figure 10 shows the platform configuration file used. As shown in the figure, each section is represented by the platform class that will be used for execution.

```
[GooglePlatform]
identity = PATH_AUTHENTICATION_KEY
region = us-east1
zone = us-east1-c
disk_image = PLATFORM_DISK_IMAGE
    [[extra]]
    is_preemptible = True

```

Figure 10. Configuration file of the platform used for transcriptome analysis. Path to the authentication key and the platform disk image is replaced by descriptive placeholders.

2.7.2. Performance results

After all the 775 transcriptome sequencing analyses were completed using CloudConductor, their runtime and cost metrics were aggregated and analyzed. The average runtime was less than 1.5 hours (~78 minutes), while the average cost per sample was less than 30 cents (~$0.28). The low average cost is explained by the usage of preemptible instance run on Google Cloud, while the short average running time is explained by the multi-level parallelism employed by CloudConductor.

From the content perspective, the results obtained using CloudConductor were very similar to the results obtained in the original study[18].

Thus, CloudConductor provides scientifically accurate results in a rapid and cost-efficient manner.

CHAPTER 3. The Whole Genome Landscape of Burkitt Lymphoma Subtypes

3.1. Introduction

Burkitt lymphoma (BL) is one of the most rapidly proliferating cancers described and is characterized by the translocation of the *MYC* gene to the immunoglobulin heavy or light chain loci resulting in its overexpression. BL has three recognized subtypes with distinct clinical presentations and geographical prevalence: sporadic, endemic and immunodeficiency-associated. BLs were first described in Africa and "endemic" BL remains the most common pediatric malignancy in sub-Saharan Africa. Sporadic BL occurs throughout the world including the Western Hemisphere. A third subtype occurs in immunodeficient patients, most commonly those with HIV infection.

Here, we applied whole genome and transcriptome sequencing of 101 tumor/normal pairs to comprehensively investigate the genomic basis of all three subtypes of BLs. In addition, the role of oncogenic drivers was functionally annotated through CRISPR screens in BL cell lines. Several genes were implicated to be involved in BL tumorigenesis including *IGLL5*, *BACH2, SIN3A* and *DNMT1*. Similar genetic profiles were observed in sporadic and immunodeficiency-associated BLs. In contrast, endemic BLs manifested more frequent mutations in *BCL7A* and BCL6 and less frequent genetic alterations in *DNMT1, SNTB2* and *CTCF*. Silencing mutations in *ID3* were a common feature of all three subtypes of BL as observed previously[19–22].

3.2. Methods

3.2.1. Sample Acquisition and Processing

All samples were deidentified and processed in accordance with a protocol approved by the Institutional Review Board at Duke University. BL tumor and paired normal samples were collected including 60 sporadic, 32 endemic and 9 HIV, totaling 101 samples, which were used in the study. The pathology diagnosis of all cases was confirmed by separate review by a panel of qualified pathologists using current WHO criteria. Whole genome sequencing was performed on all tumor normal pairs, and RNA was available for 82 of the 101 cases.

3.2.2. DNA-Sequencing Analysis

FASTQ files comprising reads from whole genome sequencing were first tested for quality using FastQC v.0.11.7 (www.bioinformatics.babraham.ac.uk/projects/fastqc). Adaptor sequences and low quality reads were trimmed using Trimmomatic[23] v.0.36. Next, alignment was performed using the BWA[24] v.0.7.15 to map reads to the human GRCh38 reference genome. PCR duplicates were marked using Picard v.2.8.2 (http://broadinstitute.github.io/picard/).

3.2.3. Somatic Variant Identification and Filtering

Somatic variant calling was performed using the Mutect2[25] from GATK4 using the default parameters. The obtained VCF files were merged and normalized using bcftools, and then annotated using Annovar[26] v.2017Jul16. Next, the variants were

filtered by first considering only “PASS” filter in at least one sample. We excluded variants that had less than three supporting reads for the alternative allele in the tumor samples and four supporting reads for the reference allele in the control samples. We removed variants found in the repetitive and low-complexity regions reported in RepeatMasker and genomic Super Dups databases, and eliminated variants with a high population allele frequency (>0.01) reported in ExAC, gnomAD exome and gnomAD genome databases. Finally, we filtered out variants that have an average Median Base Quality and average Median Mapping Quality lower than 10.

3.2.4. RNA-Sequencing Analysis

FASTQ files comprising reads from RNA sequencing were first tested for quality using FastQC v.0.11.7. Adaptor sequences and low quality reads were trimmed using Trimmomatic[23] v.0.36. We used STAR[27] v.2.5.1a aligner to map reads to the human transcriptome. PCR duplicates were marked using Picard v.2.8.2. The transcript abundances were calculated using RSEM v1.3.0[28]. The final output was a matrix of FPKM units per transcript. Custom scripts were used to perform differential analysis. Gene set enrichment analysis was performed using GSEA[29]. All statistical analyses and plots were generated using R v3.4.4.

3.2.5. Copy Number Variation Analysis

Copy Number Variation (CNV) analysis was performed on whole genome sequencing (n=101, tumor-normal pairs) samples. We used the copy number workflow from the GATK4 toolkit (v4.0.1.1) to make CNV calls as described in

https://software.broadinstitute.org/gatk/documentation/article?id=11682. The tool uses a standardized and denoised read-depth method to detect CNVs and produce copy number ratios (CNR), and a Gaussian-kernel binary-segmentation algorithm to detect copy number segments for each sample.

3.2.6. Identification of clusters of genomic alterations

We implemented a custom Python script to identify genomic clusters, i.e. regions in the genome with at least 4 variants and where the variants are at most 200 base pairs apart from adjacent variants. For each identified genomic cluster, we computed the number of unique samples that have variants in the specific cluster. We manually curated the clusters by analyzing their location in the genome and the quality of the variants involved, and visual tests using IGV. Next, using bedtools, we intersected the filtered cluster regions with genes location and their promoters (2000 base pairs before the gene) using data obtained from UCSC Table Browser.

3.2.7. Translocation Identification

We identified translocations using Delly2[30] and querying discordant reads, i.e. read pairs that map to two different chromosomes. We clustered the discordant reads based on the positions on both chromosomes. We considered translocations that have a cluster of at least two pairs of discordant reads with good mapping quality (>30).

3.2.8. EBV Identification

We identified the presence of EBV virus using Diamond v0.9.18[31]. EBV subtype calls were made for EBV+ samples by determining the proportion of Type-1 (B95-8) vs. type-2 EBNA-2 (AG876) sequences among unmapped reads. All cases were verified using *in situ* hybridization for *EBER* where material was available.

We obtained the unaligned read pairs from the tumor and normal samples and performed Diamond BLASTX on the Non-redundant (NR) protein sequences database. The output format of Diamond alignment was set for taxonomic classification ("-f 102"). Samples were considered EBV-positive if at least 5,000 reads or >10% of unmapped reads were classified as *Lymphocryptovirus* at the genus level (NCBI tax ids: 10375, 10376, 10377). EBV status calls were not made for samples with fewer than 10,000 input unmapped reads.

EBV subtype calls were made for EBV+ samples by determining the proportion of Type-1 (B95-8) vs. type-2 EBNA-2 (AG876) sequences among unmapped reads. Briefly, representative nucleotide sequences were downloaded in FASTA format from UniProt (Type 1: P12978, Type 2: Q69022). As above, reads were aligned to a custom diamond database containing only representative sequences from each EBV subtype (diamond makedb -in <ebna2.fa>) with the option –max-target-seqs set to 1 to return only the top alignment. Alignments were filtered to include only significant hits which were defined as >35 amino acids in length, >50% amino acid identity, and with a maximum e-value of 1.0e-5. After filtering, a custom R-script was then used to determine

the number of reads in each sample mapping to each EBNA-2 gene variant. A subtype call was made if >99% of reads mapping to EBNA-2 mapped to a single subtype. All cases were verified using in situ hybridization for EBER where material was available.

3.2.9. CRISPR Screening

Three BL cell lines (BJAB, BL41 and Jijoye) were subjected to a CRISPR screen using the GeCKO v2.0 CRISPR library system containing over 120,000 guide RNAs (sgRNAs) targeting over 19,000 genes. Infected cell lines were harvested 3 days post transduction (early) and 3 weeks post transduction (late), DNA isolated, and targeted sequencing of the guide RNAs was performed.

Lentiviral particles were transduced into each cell line in triplicate, and harvested 3 days post transduction (early) and 3 weeks post transduction (late). In each case, DNA was isolated and targeted sequencing of the guide RNA sequences was performed.

High throughput Illumina sequencing of sgRNA libraries amplified from plasmid sequences was used to determine sgRNA abundance for populations at each time-point. The median raw sequencing depth across sequencing libraries was 33,951,761 reads corresponding to 282 reads/sgRNA for a given library. sgRNA preprocessing, QC, quantification and computing gene knockout scores was performed similar to our previous publication. Subsequent sgRNA abundance was assessed and compared between the two time points. Highly abundant sgRNAs were predicted to target tumor suppressor genes, whereas less abundant sgRNAs were predicted oncogenic.

3.3. Results

3.3.1. Somatic genomic alterations identified from whole genome sequencing of BL

We performed whole genome sequencing of 101 BL tumor samples and their matched controls. This set of 101 BL patients included 60 sporadic, 9 HIV-associated and 32 endemic BL cases. These tumor-normal pairs were subjected to whole genome sequencing, targeting a mean genome-wide coverage of 75X. In all, we identified over 2 million variants from these cases including 228,010 somatic variants[25] that were analyzed further.

We developed an approach to identify the genomic regions comprising a high density of somatic variants representing "clusters" of alterations occurring anywhere in the BL genome. We identified 220 such genomic clusters that included both exonic and noncoding regions, which included at least 4 variants from any disease subtype within 200bp of each other, and affected at least 3 samples. We found 620 filtered somatic mutations (26.87%) that were associated with AID activity (i.e. mutation was present within 3bp of an AID recognition site WRCY). We integrated these findings into a model[32] that we have applied previously[18] to identify genetic drivers.

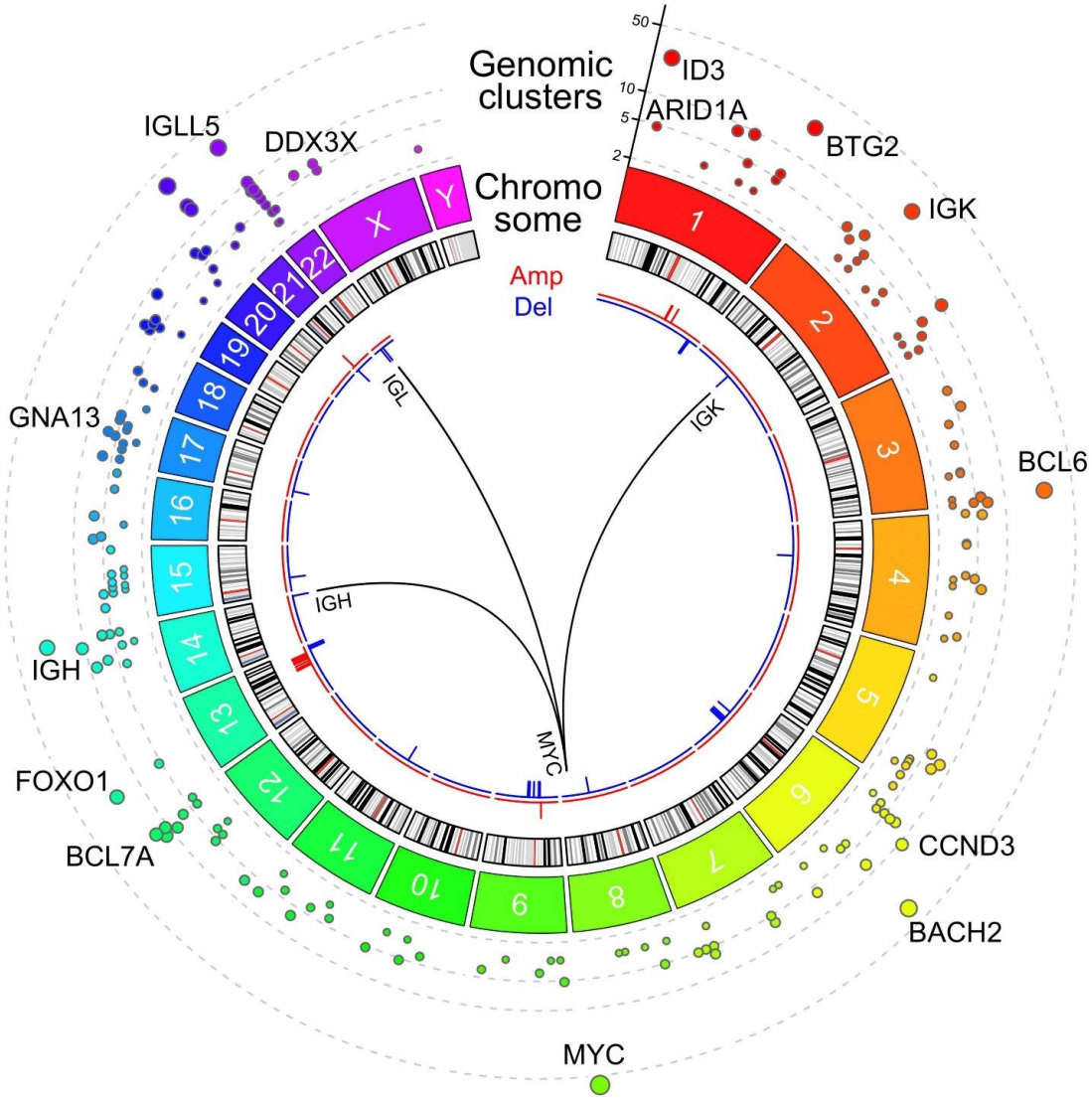


Figure 11. Whole genome sequencing of Burkitt lymphoma (N=101). Tracks, from outside to inside, on the Circos plot: genomic clusters containing at least four somatic variants in at least three samples (y-axis shows the unique samples count) scaled by the mutated sample counts and copy number variations represented as amplification (red) and deletions (blue). The internal arches represent the three translocations involved in the disease between *MYC* gene (chr8) and immunoglobulin genes *IGH* (chr14), *IGK* (chr2) and *IGL* (chr22).

Figure 11 shows a Circos diagram[33] of the human genome with chromosomes and their respective ideograms. Genomic clusters are shown in the outer track as circles with the radius representing the number of affected samples. Notably, *MYC*, *IGLL5*, *BACH2*, *ID3* and *BTG2* were identified as prominent genomic clusters with multiple somatic events in both coding and noncoding regions of these genes. The largest genomic cluster was identified in the oncogene *MYC*, containing 242 somatic events from 59 tumor samples in a 4011 bp window, consistent with the well-described role of *MYC*.

We also identified structural variations including somatic, recurrent copy number alterations (both amplifications and deletions; frequency in at least 3 samples) as well as chromosomal translocations. Chromosomal translocations involving *MYC* were also assessed in all cases as part of the clinical diagnosis. As expected, *MYC* translocations with *IGH*, *IGK* or *IGL* were observed in all cases (Figure 11). In addition, next generation sequencing identified a novel translocation between *MYC* (chromosome 8q24) and *ACTB* (chromosome 7p22). Through clustering discordant reads (i.e. read pairs whose mates map on different chromosomes), we identified a novel translocation between *MYC* (chr8) and *ACTB* (chr7) (Figure 12A). We used density based spectral clustering[34] to cluster the discordant reads using a Euclidean distance metric (max distance of 300 bases). The breakpoints were identified using chimeric reads in *MYC* and *ACTB* genes. We then validated the presence of this translocation using Sanger sequencing around the breakpoint region (Figure 12B). This translocation was confirmed using Sanger sequencing. Somatic copy number alterations were identified by comparing tumor

samples to a pool of normal samples taken from the same 101 patients. We identified a high prevalence of copy number alterations in *IGLL5* (deletions) and *MCL1* (gains).

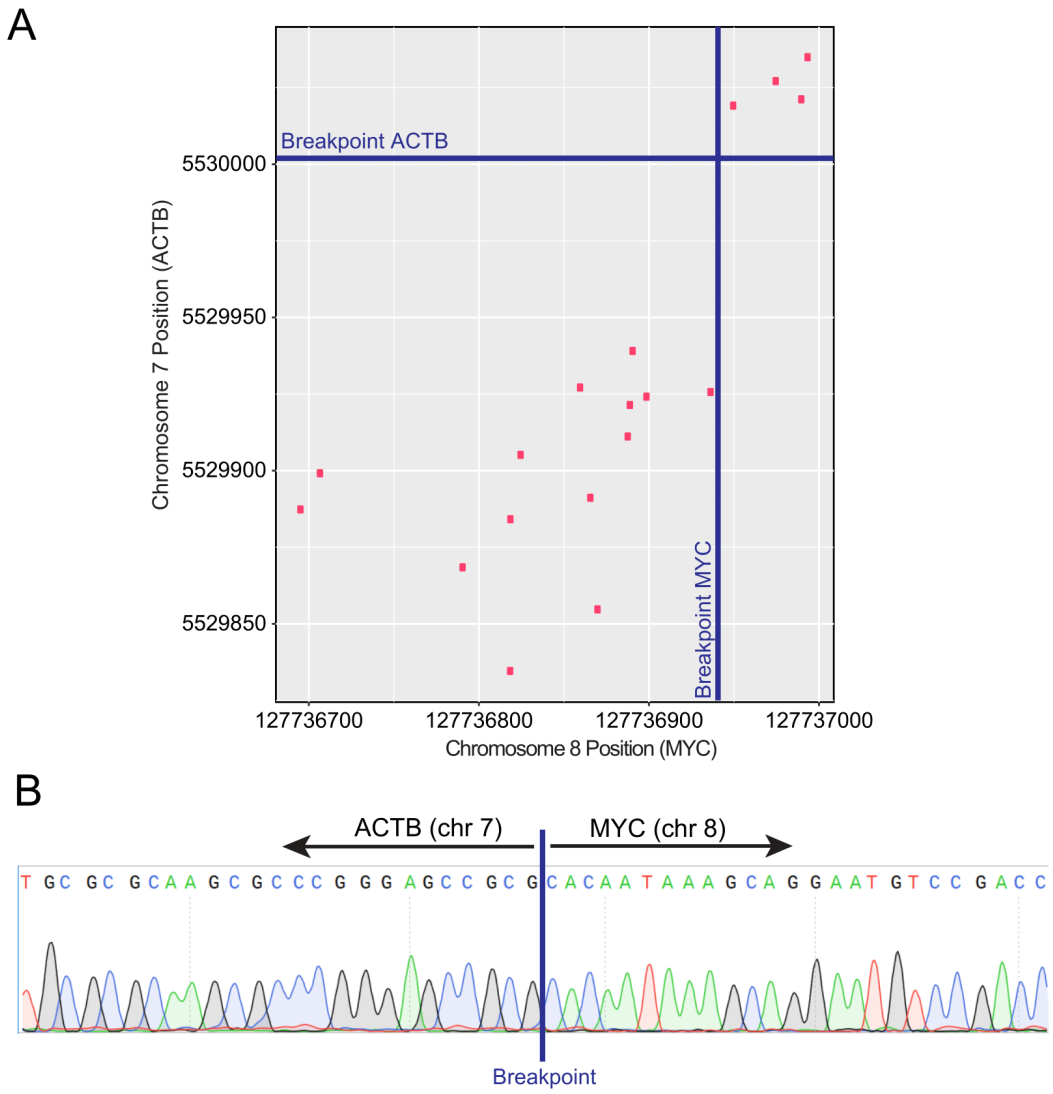


Figure 12. Validation of translocation t(8,7) – *MYC* and *ACTB*. (A) Scatter plot shows discordant reads that support the translocation, with breakpoints shown as a blue line. (B) Sanger sequencing plot shows ±25 base pairs around the breakpoint region (shown as a blue line).

3.3.2. Landscape of driver genes and their association with subtypes

In all, we identified a total of 72 driver genes in BL using approaches that we have developed previously[18]. The heatmap of genetic alterations shows driver genes found in at least 15 BL patients ordered by their subtype and mutational frequency (Figure 13A).

The patterns of mutations and structural alterations provided important clues to the oncogenic behavior of these driver genes. Variant stem plots for the top mutated driver genes with a high proportion of non-coding variants are shown in Figure 14. These plots show an enrichment of non-coding variants in the driver genes and highlight the importance of studying whole genomes in Burkitt lymphoma. Variants are plotted across the genomic coordinates (x-axis) with their height showing the number of mutated Burkitt lymphoma samples[35]. The variants are colored based on their functional annotation (missense, truncating or non-coding). The colored bars along the x-axis show the promoter and the exons.

In addition to translocations, somatic mutations also occurred commonly in *MYC* as missense alterations sparing the HLH domain. *BCL6* and *MCL1* show a similar enrichment of missense mutations and amplifications consistent with a gain-of-function of these oncogenes. On the other hand, *DDX3X*, *ARID1A* and *ID3* manifested frequent truncating mutations and deletions consistent with a loss-of-function of these tumor suppressors.

Non-coding mutations were a major feature of the mutational landscape of BLs, with a majority of the driver genes showing multiple non-coding events. Over 90% of the samples had non-coding genetic alterations in the driver genes. The high prevalence of non-coding mutations even in known driver genes underscores the importance of whole genome sequencing to characterize the driver events in BL. Interestingly, we observed that many non-coding variants associated with driver genes occurred in the promoter region or in the first intron. For 10 of the 72 driver genes, specifically *IGLL5*, *BACH2*, *BTG2*, *BCL6*, *BCL7A*, *TCL1A*, *IRF8*, *CXCR4*, *ZFP36L1*, and *S1PR2*, we identified patterns consistent with somatic hypermutation[36].

Separately, we examined the profile of driver events in the context of whole genome sequencing of DLBCL[37] and found strong overlap between both coding and non-coding events in GCB DLBCL including those affecting *BCL6*, *DNMT1*, *BCL7A*, *IRF8* and *FOXO1* ($p<0.01$, chi-squared test), implicating shared lineage from germinal center B cells as a major influence in the acquisition of driver events[38].

The Epstein-Barr virus (EBV) is known to be a critical contributor to the pathogenesis of BLs. Not unexpectedly, EBV+ cases were significantly enriched in the endemic subtype (81% EBV+, $p<0.001$, chi-squared test). Among the sporadic BL cases, 20% were EBV+ (Figure 13A).

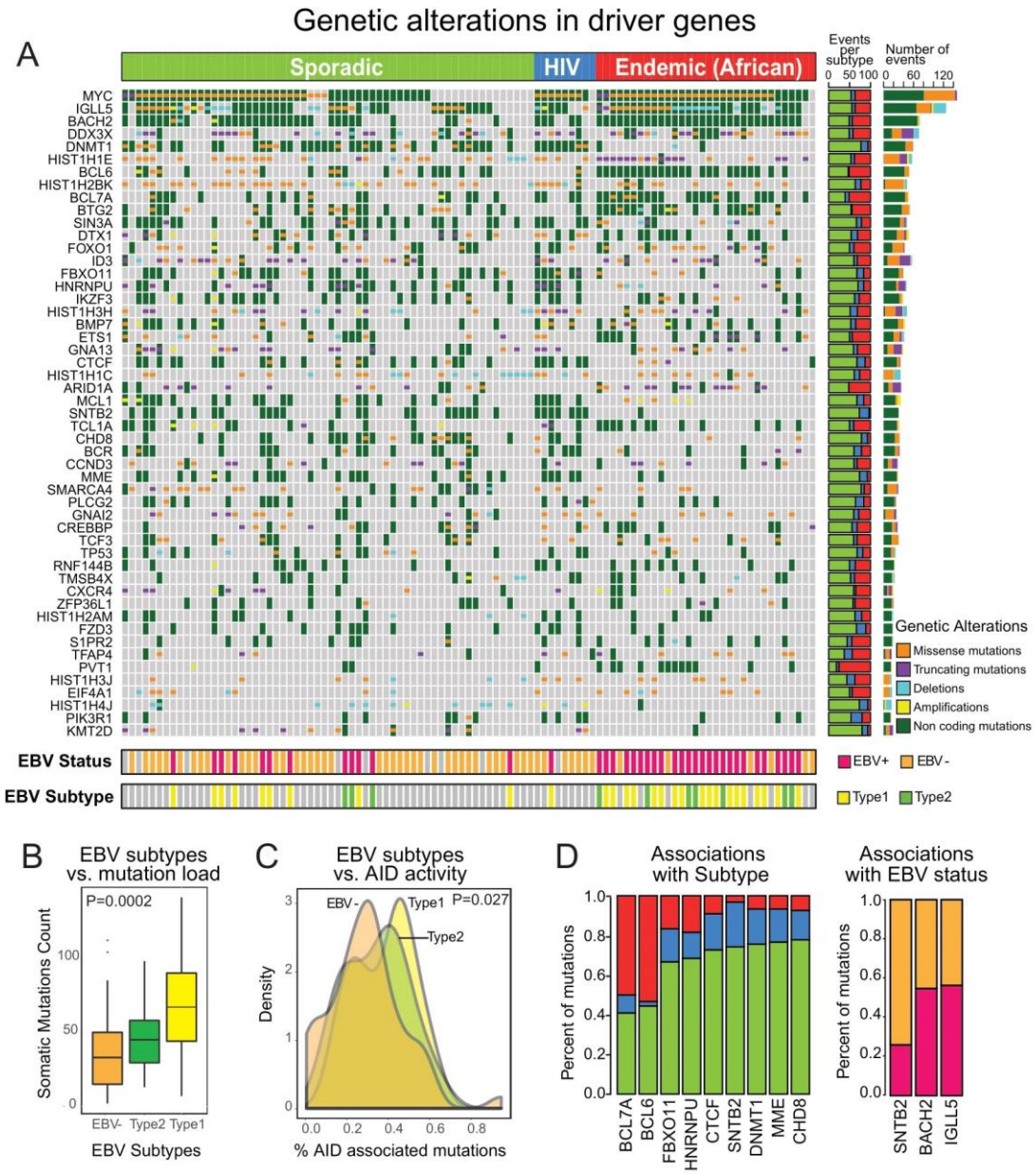


Figure 13. Landscape of genetic drivers in Burkitt lymphoma subtypes. (A) Heatmap of genetic alterations across Burkitt lymphomas for the driver genes mutated in at least 15 samples. The barplot on the right shows the sample counts for each of the driver genes. (B) Boxplots showing the differential mutation load across the EBV subtypes. (C) Distribution of the proportion of AID-associated mutations for EBV status. (D) Barplots showing genetic alterations significantly associated with Burkitt lymphoma subtypes (left) and EBV status (right).

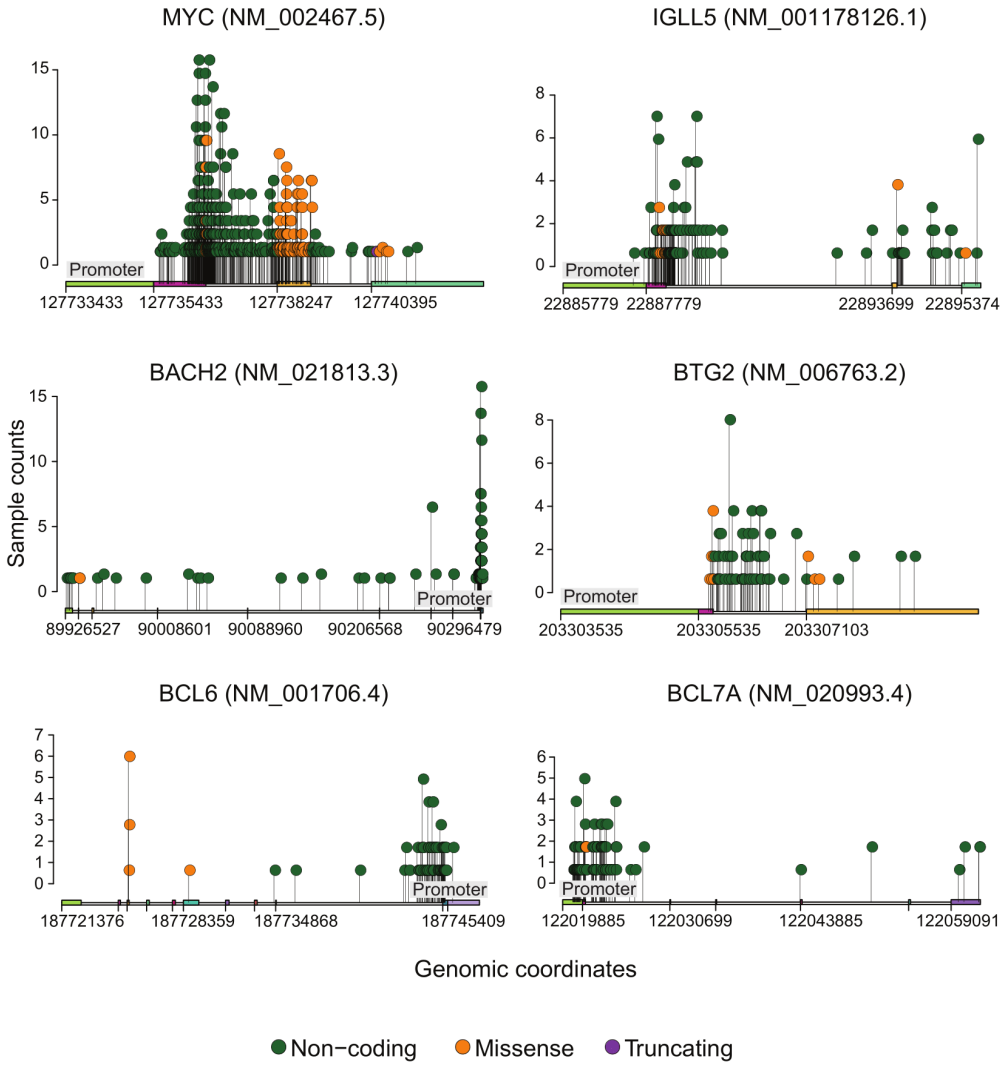


Figure 14. Distribution of somatic variants in driver genes. Variant-stem plots for selected driver genes show the distribution of somatic variants across the genomic coordinates on the x-axis and sample counts for each of the variants on the y-axis. The colored bars on the x-axis show the promoter and the exons. The promoter regions are also labeled. Variants are colored based on their function.

We further analyzed the associated between somatic mutation load and EBV status. We found that EBV+ patients had a significantly higher mutation load compared to EBV- cases ($p<10^{-5}$, Wilcoxon test). This association was also observed when we

stratified the patients by BL subtype, indicating a strong nexus between EBV infection and downstream genetic events. We further examined the effects of EBV subtype and found that EBV+ Type1 patients had a significantly higher mutation load compared to EBV+ Type2 tumors (Figure 13B, p=0.0002, ANOVA).

The observed high mutational burden led us to examine the relationship between AID-associated mutations and EBV subtypes. EBV+ BLs were associated with a higher proportion of AID-associated mutations compared to EBV- BLs (Figure 13C, p=0.027, ANOVA).

We analyzed the mutational spectrum that contributed to the somatic mutations (Figure 15**)** and identified three mutational signatures that are broadly associated with BLs. We compared these mutational signatures to the database of published signatures[39]. EBV- endemic BLs were associated with a strand bias for C>T mutations, while EBV+ endemic BLs were associated with dysregulation of AID activity during somatic hypermutation. On the other hand, somatic mutations in sporadic and HIV BL samples resembled a signature correlated with a transcriptional strand bias for C>A mutations[39]. We further identified genetic alterations that were associated with one or more BL clinical subtype or EBV status (Figure 13D). We found that *BCL7A* and *BCL6* genetic events were enriched in the endemic subtype, while *DNMT1*, *SNTB2* and *CTCF* mutations occurred more frequently in HIV and sporadic cases ($p<0.05$). *SNTB2* mutations were associated with EBV- cases, *IGLL5* and *BACH2* mutations were associated with EBV+ cases ($p<0.05$).

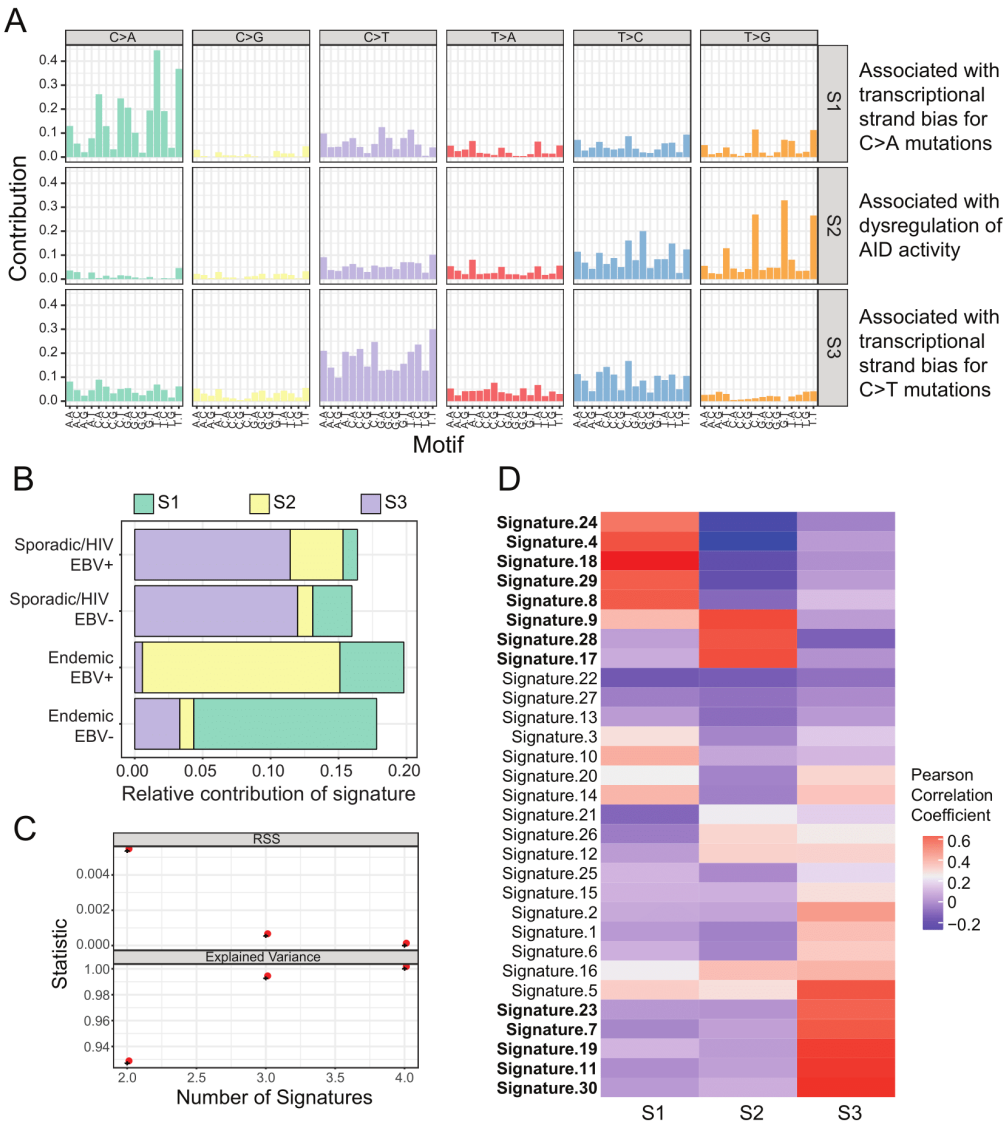


Figure 15. Mutational signatures analysis. (A) Mutational spectrum of the three mutational signatures computed from the somatic variants. (B) Barplots representing the contribution of the somatic mutations signatures. (C) Scatter plots presenting the residual sum of squares and the explained variance of NMF decomposition with different numbers of signatures. (D) Heatmap shows Pearson correlation between the three signatures and the annotated mutation signatures from COSMIC database.

Interestingly, both BCL7A and BCL6 are more frequently mutated in the endemic subtype (Figure 13D) and are targeted by the somatic hypermutation[36], implicating the connection between EBV infection and dysregulation of AID-activity.

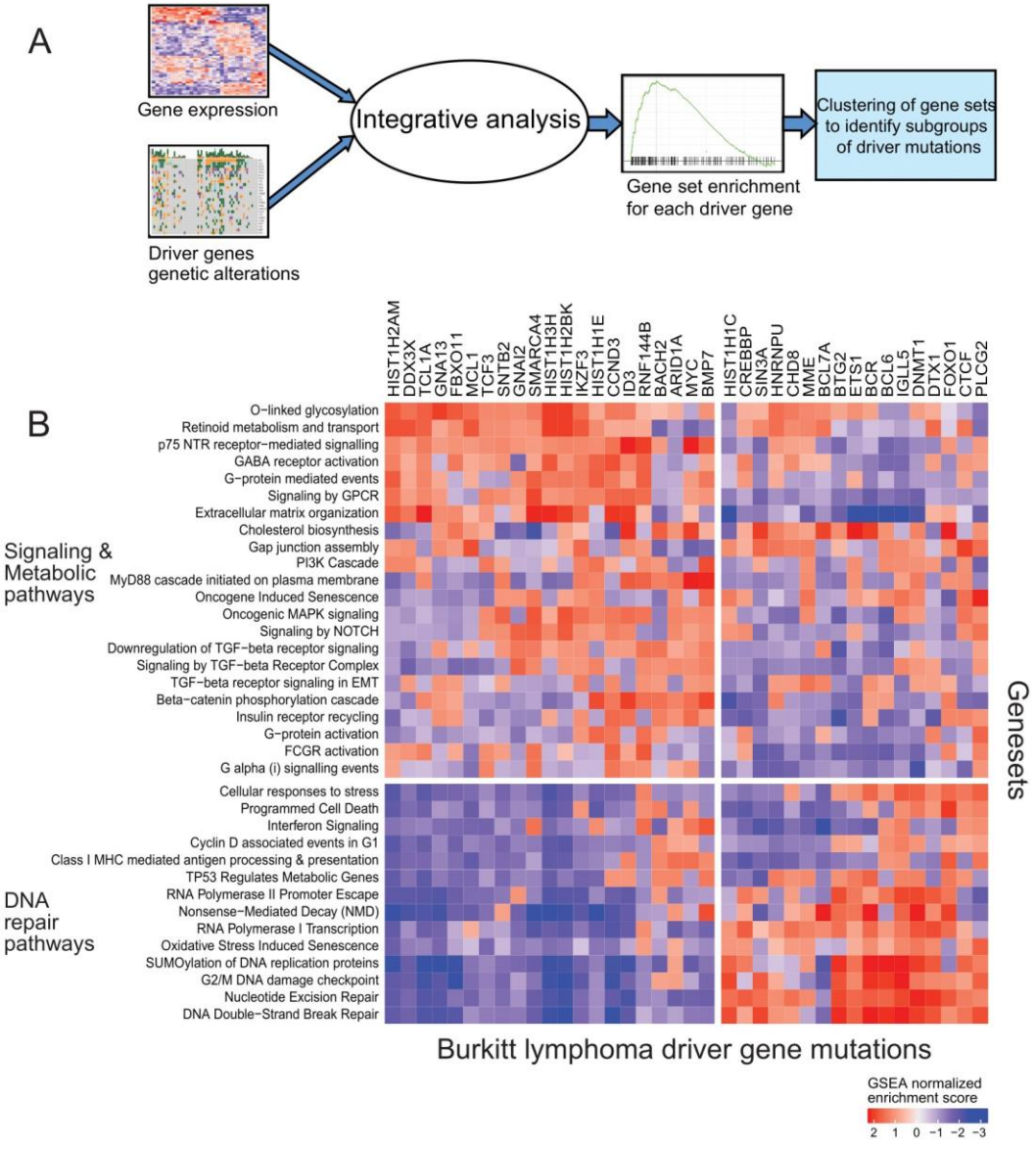


Figure 16. Expression patterns across Burkitt lymphoma. (A) Schematic showing the integrative analysis of driver gene mutations and gene expression profiles for Burkitt lymphoma. (B) Heatmap showing enrichment scores for genesets associated with genetic

alterations in each of the driver genes shows two clusters of genesets: signaling and metabolic pathways, and DNA repair.

3.3.3. Expression patterns associated with driver mutations in BL

We examined the downstream changes in gene expression associated with the driver mutations. To identify differentially expressed genes and pathways associated with genetic alterations in each of the driver genes, we performed logistic regression modeling by using BL subtype as a covariate to account for the differences between the subtypes. We next performed gene set enrichment using described gene ontologies[29,40] with odds ratios (logistic regression coefficients) as the ranked metric for enrichment analysis (Figure 16A).

The normalized enrichment scores from gene set enrichment were then visualized as a heatmap for each of the driver genes (Figure 16B). We found that the gene sets clustered into two main groups involving (i) signaling and metabolic pathways, and (ii) DNA repair. Signaling pathways, including G alpha signaling and G-protein mediated events, were enriched in *GNA13* driver mutations. Mutations in *DNMT1* and *BCL6* were associated with DNA repair pathways.

3.3.4. Comparison of genetic alterations and gene expression of BL and DLBCL

BL and DLBCL are both aggressive B cell lymphomas with overlapping morphology, immunophenotype and genetics. This overlap can make the distinction of BL and DLBCL challenging. Given the vast differences in treatment of BL and DLBCL,

the distinction is clinically important. We therefore compared the genetic alterations and gene expression of BL and DLBCL using publicly available data from 1001 DLBCLs.[18]

We first compared the mutational profiles of BL and DLBCL. The overlap of the driver genes is shown as a Venn diagram (Figure 17A) and as a barplot showing mutational frequencies for the top driver genes in BL and DLBCL (Figure 17B). Roughly 40% of the BL driver genes (31 out of 72) are shared with DLBCL, while the remaining 60% are altered predominantly in BL, suggesting divergence in mutational patterns for these two lymphomas.

We further compared the expression profiles of BLs and DLBCLs. We first identified differential genes for each of the BL subtypes and compared these to the cell of origin-based DLBCL subtypes, activated B cell like (ABC DLBCL) and germinal center B cell like (GCB DLBCL). We identified a total of 2,563 differentially expressed genes that distinguished at least two BL subtypes from DLBCL (Figure 17C). In general, we observed that the gene expression patterns of DLBCL were more heterogeneous than BLs, even when BL subtype differences were taken into account. As expected, GCB DLBCLs were closer in gene expression to BLs compared to ABC DLBCLs.

We plotted the gene set enrichment analysis results separately for differentially expressed genes associated with ABC DLBCL and GCB DLBCL, and by the direction of the effect (up-regulated or down-regulated). These significantly enriched gene sets are shown in Figure 17D. As expected, MYC target genes were more strongly associated with BL compared to DLBCL. Additionally, cell cycle and MTORC1 signaling had

higher expression in BL compared to DLBCL, while apoptosis and JAK-STAT pathways were more highly expressed in DLBCL.

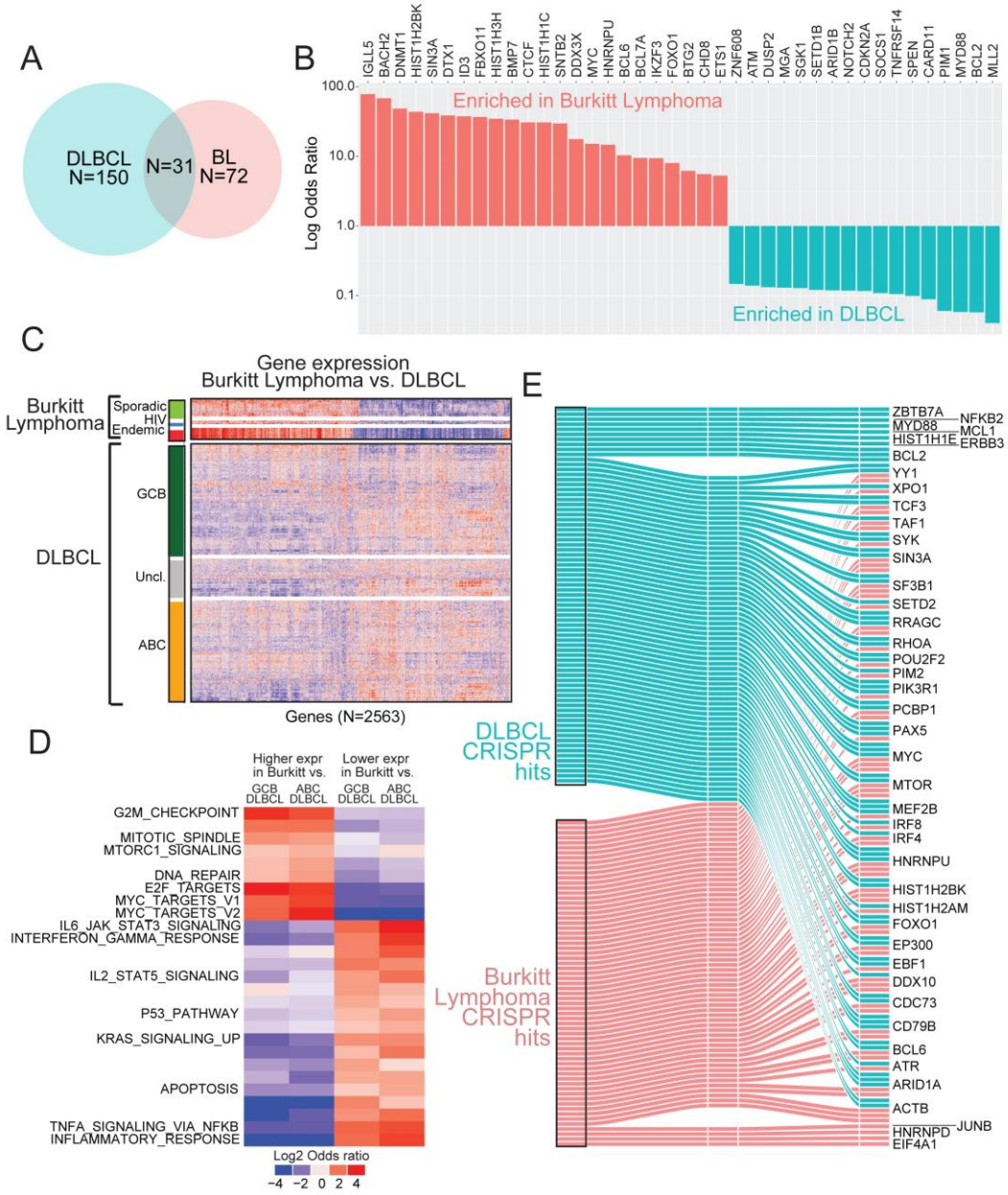


Figure 17. Comparison of Burkitt lymphoma to DLBCL. (A) Comparison of mutational driver in BL vs. DLBCL are shown in a Venn diagram. (B) Barplot showing contrasting mutational frequencies. (C) Heatmap of differential genes is shown for BL

and DLBCL subgroups. (D) Heatmap of differential gene sets for BL vs. ABC, GCB DLBCL. (E) Essential genes in BL and DLBCL identified by CRISPR KO screen. Essential genes are grouped as BL-specific, DLBCL-specific, or shared (effects in both) effects upon knockout.

3.3.5. CRISPR screening to functionally annotate BL driver genes

In order to better understand the functional effects of the identified driver genes, we utilized a genome-wide human sgRNA library containing over 120,000 sgRNAs targeting 19,050 protein coding genes in three BL cell lines (BJAB, BL41, Jijoye)[9] and compared it to three DLBCL cell lines (SUDHL4, Ly3, HBL1). We transduced three replicate populations from each cell line with the complete sgRNA library and sequenced replicate populations at two time-points (1 and 14 population doublings post transduction) to observe changes in sgRNA frequency over time.

Genes whose knockout led to depletion at the late time point were deemed to be "essential genes" similar to that described previously[41]. Broadly, BL-essential genes were observed to be enriched for cancer-related functions including MYC targets, cell cycle and DNA repair, as well as other critical cellular processes including ribosome biosynthesis, translation and the metabolism of RNA and mRNA (Figure 18). We noted that 15 BL driver genes were among these essential genes. Essential genes shared among BL and DLBCL included *MYC*, *BCL6*, and *ARID1A* (Figure 17E). We observed fewer BL-specific essential genes than DLBCL-specific genes, likely owing to the greater genetic heterogeneity of DLBCLs compared to BL.

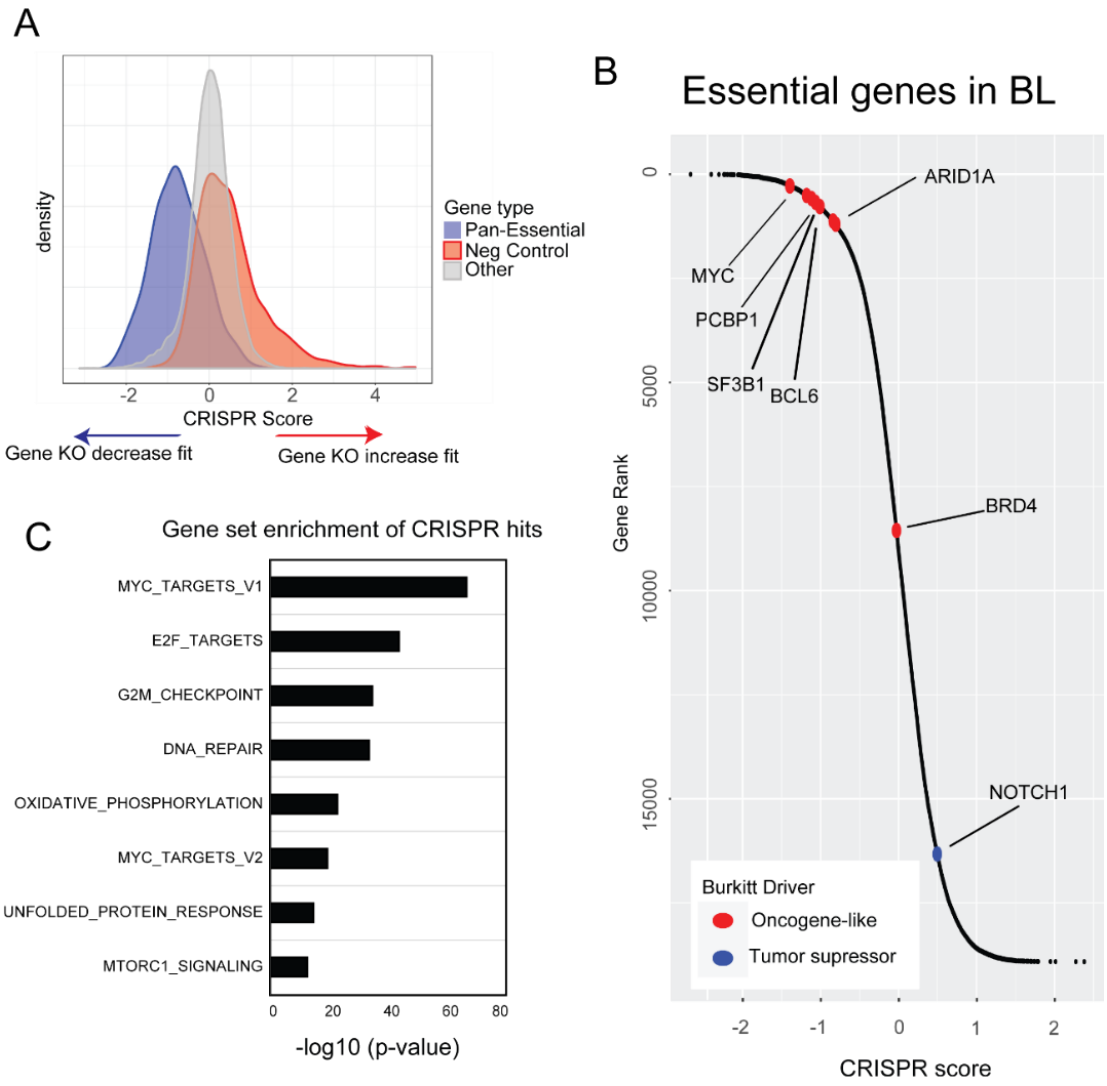


Figure 18. Essential genes identified using CRISPR screens in BL. (A) Histogram of CRISPR scores shows that pan-essential genes have lower scores compared to negative controls. (B) Waterfall plot showing genes ranked by CRISPR score. Essential genes which are also BL drivers are highlighted. (C) Gene set enrichment performed on CRISPR hits using Msigdb Hallmarks gene sets are shown as a barplot.

3.4. Discussion

The three subtypes of BL have distinct clinical presentations. Through whole genome and transcriptome sequencing of all three subtypes, we identified 72 driver genes

in all three subtypes, with both coding and non-coding mutations. Interestingly, the sporadic and HIV subtypes were much more closely related to each other, at both the genetic and transcriptional level, than the endemic subtype. Notably, whole genome sequencing identified a number of non-coding mutations in nearly all genetic drivers in BL, indicating that whole genome sequencing greatly expands our understanding of genetic events that converge in Burkitt lymphoma subtypes.

Additionally, CRISPR screening in BL cell lines indicated both oncogenes and tumor suppressor genes that have a role in cell proliferation. *ID3* is one of the most commonly silenced genes in all three BL subtypes.

Our data has several clinical implications. First, the genomic landscape of BL identifies expression and genetic markers that enable the sometimes challenging, but clinically important distinction of BL from DLBCL. Evolving technologies might enable their direct measurement through DNA and RNA sequencing, obviating the need for a number of individual markers currently needed to distinguish DLBCLs in clinical pathology. Second, our data indicate that the clinically distinct BL subgroups arise from highly shared genetic origins. Thus, therapeutic approaches and clinical trials that are relevant to one subgroup are relevant to the other subgroups. For instance, a clinical trial conducted in sporadic BL in the Western Hemisphere is relevant to HIV patients with BL as well as BL patients in Sub Saharan Africa, even though the latter groups may be logistically difficult to enroll in a trial. Finally, our data support the continued evaluation of EBV status in BL patients.

Our study thus provides a comprehensive examination of the genomic alterations in BL subtypes and provides a rich starting point for viable models of the disease and understanding its biology.

CHAPTER 4. An Automated Cloud-Based Infrastructure

4.1. Introduction

Although a workflow management system successfully automates and simplifies the processing of running a pipeline, managing the input and output data of each pipeline and reporting the status of a running pipeline can become very demanding when processing a large cohort of samples. Additionally, manually managing the computational resources on which the workflows are running can lead to inefficient resource allocation and manipulation. In order to use a workflow management system on a large number of samples, additional layers of infrastructure are needed. We developed such an infrastructure to automate the process of running such multi-sample workflows. As presented in the introduction of this thesis, there are five large components that are interconnected in an automated infrastructure.

We have implemented an infrastructure that uses Kubernetes (https://kubernetes.io/) an open-source application to automate the deployment, scaling and management of the analysis performed by CloudConductor. Figure 19 presents a general schematic of the infrastructure. The users interact directly with the website, which is outward facing component of this system. All the sample, sequencing and analysis data is available on the website for the user to input and/or access. Besides the workflow management system, all the components are connected through the database,

which serves to orchestrate communication between the components of the system. The workflow management system is the only component that is isolated from the database, as its input and output information are being controlled by the managing daemon.

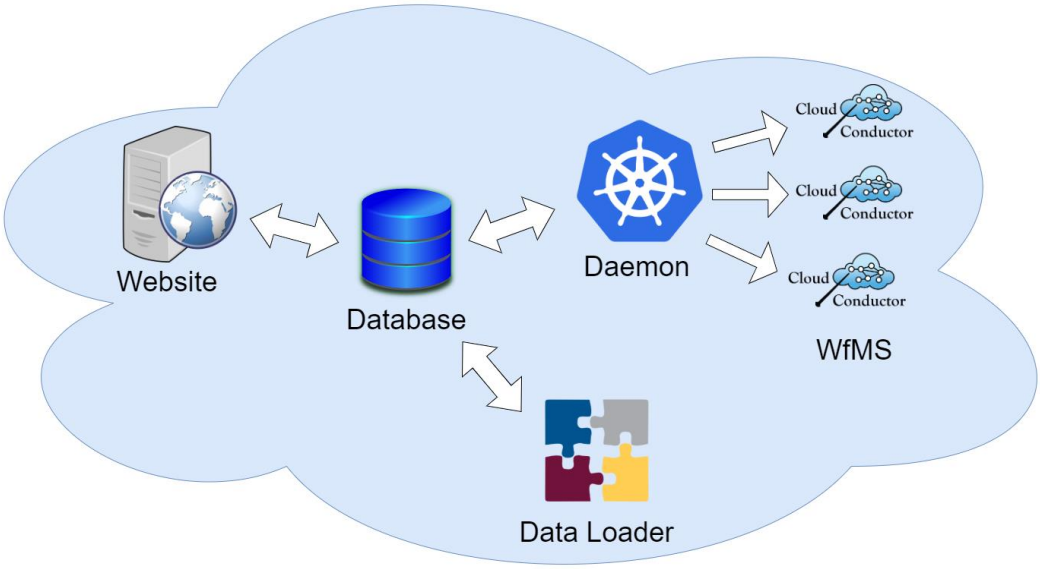


Figure 19. General design of the automated infrastructure. All infrastructure components are present on the cloud platform. The components are interconnected with the database, except the WfMS, which is launched independently by the daemon.

The process begins with the arrival of sequencing data from the sequencing facility. Using the website, the user, with a single click can update the database with the new analysis request using the data loader. The managing daemon, which continuously monitors the database, automatically sees the new analysis request and launches CloudConductor on the new input data. When the analysis is complete, CloudConductor returns the analysis statistics to the storage system and the managing daemon populates the database with the new analysis metadata and output paths. Overall, this system allows the users to access all the analyses that they have done in the past, with the accompanying

statistics and output data. The users can also rerun the analyses in the future if needed. Thus, using all these components, the infrastructure allows the whole process to be rapid, automated and reproducible.

In this chapter, I will present the five components of the automated infrastructure and how we implemented them followed by an example of how the whole infrastructure can be used to execute a complex pipeline.

4.2. Components

4.2.1. The website

The first component necessary for the automated infrastructure is the website. To implement the website, we used the Django framework. The visual elements of the website, such as buttons, were implemented using the Bootstrap library. Overall, the implementation of the website structure is directly dependent on the intended bioinformatics application. For example, for genomics applications, the website should display information about sequencing data and how the sequencing libraries have been prepared.

The main functional purpose of the website is to provide access to the infrastructure. The user would use the website to obtain information regarding research-related elements, such as samples or sequencing data, and analysis-related data from the database component. Additionally, the website provides the functionality for the user to

request new analyses. These requests are stored in the database to be later parsed by the data loader.

Our genomics work requires five main elements on the website. These include sample data such as diagnosis and location of the tissue samples. Additionally, we provide both the sequencing submissions metadata and the information of the sequencing preparation. Finally, we incorporated information about the analyses performed on the sequencing submissions and the possibility for the user to explore other genomic data generated by the analyses, such as the identified variants and alignment statistics. Figure 20 shows the interface of the website we created with all the described elements in the header.

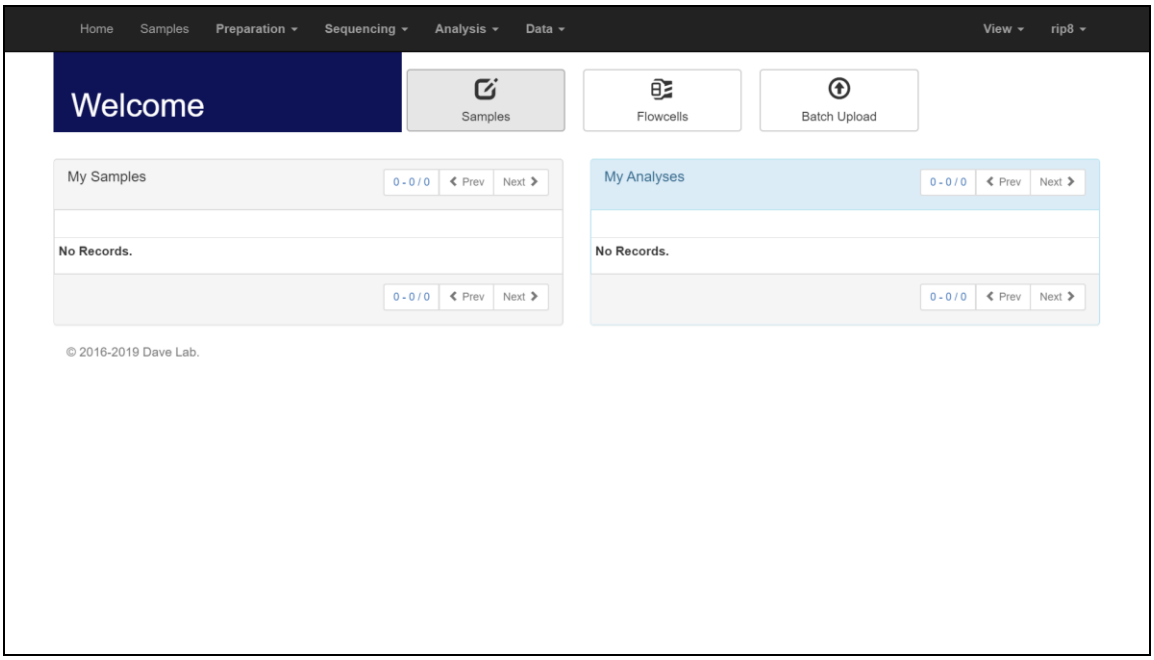


Figure 20. Overview of the website interface. All biological data types can be found in the header of the webpage.

The website was created as a designated instance on the cloud that runs the webserver using Apache. Finally, the website domain directs to the external IP address of the configured instance.

Thus, the website provides a method of communication between the infrastructure and the user.

4.2.2. The database

The second component of the automated infrastructure is the database. As with the website, the design of the database depends on the specific bioinformatics application for which it will be used. We used a genomics schema implemented in MySQL. A general overview of database schema is shown in Figure 21. The database comprises two sections that are connected by one table.

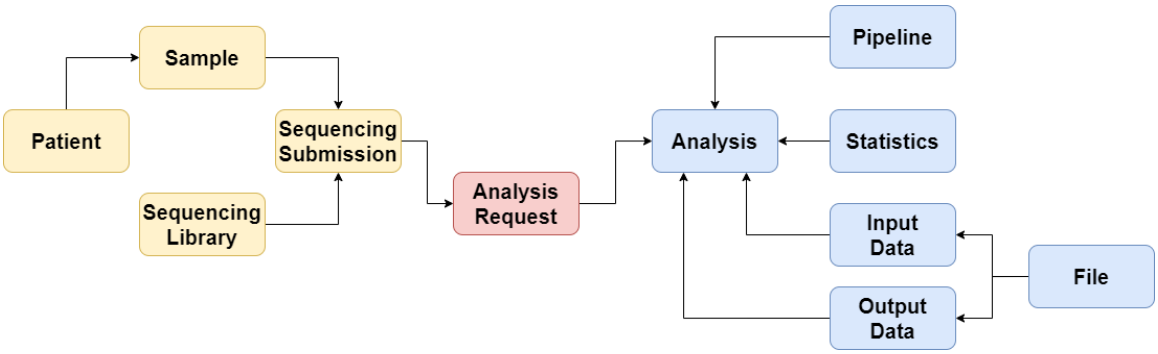


Figure 21. Database schema for genomics data types. Tables are separated in two sections: the yellow section containing the biological input information and the blue section containing the analysis metadata and results. The table colored in red represents the analysis requests that make the connection between the two sections.

The first section is represented by the clinical and biological information tables. The first section of the database is strongly related to the specific bioinformatics. In our

case, this section contains information regarding the patients (e.g. diagnosis, tissue type, etc.), samples, sequencing submissions and paths to the sequencing files.

The second section is defined by the analysis-related tables. These tables contain all the metadata information of the analyses performed, the pipeline definitions and the analysis output data. An analysis record contains references to the processing pipeline, the input data, the produced output data and the runtime statistics. The runtime statistics include information about the total running cost and biological statistics returned by the pipeline modules.

The two sections are connected by a table that stores the analysis requests generated by the user from the website. This table contains the input data requested to be analyzed and the type of analysis to be performed on it. The most essential tables in the presented schema are the ones defining the sequencing submissions and the analysis performed.

Finally, it is important to note that files are represented in all tables only as paths to an external storage system and not by their actual file data, thus reducing the database size. The only exceptions for this file representation are the input configuration files for CloudConductor. These configurations files are small in size and their retrieval time is reduced dramatically when stored directly in the database as opposed to storing them on another external system.

4.2.3. The data loader

The third component is the data loader. We implemented the loader as a back-end Python module that translates the sample, sequencing and analysis data from the database into a sample sheet, i.e. the input required for CloudConductor.

The main task of the data loader is to identify the data for all the input keys defined in the sample sheet and populate them accordingly. All input keys can be obtained from the database and they represent sample, sequencing and analysis information. In most of the cases, retrieval of the sample and sequencing information is trivial, while obtaining keys from the analysis is more complex. To identify the necessary input keys in the output of the provided analyses, the data loader first generates the analysis graph that includes the actual input sequencing submissions. Next, the loader locates the necessary keys in the output keys of the analysis by traversing the analysis graph in a breath-first search manner. Although this is a general overview of how the data loader works, there are a number of exceptions/special cases that appear during this process.

The data loader has been implemented to automate the identification and solution to such special cases. An example of a special case is the not infrequently encountered situation that requires the resequencing the same sample multiple times. When performing analysis on this sample, it would be ideal to retrieve all the previous data that was obtained for that sample. In this case, when a user requests the analysis of one of the

sequencing files, the data loader automatically identifies and retrieves the other resequencing files.

Another example of a special case is the selection of a specific output key from an analysis. There are numerous situations when an analysis generates multiple files of the same type. For example, an analysis can return many alignment files that have different filtering levels and different analysis steps applied on them. When requested to obtain an alignment file from such an analysis, the loader has to distinguish them and allow the user to select the preferred one. To facilitate this, we have implemented a "selection rule" feature in the data loader. Using this feature, the user is able to specify keywords that differentiate the conflicting input keys and specify a rule that chooses which output key to be selected.

To connect the data loader to the database we used SQLAlchemy as the object-relational mapper (ORM). An ORM is a program that represents records from a database schema into a set of connected objects. Using SQLAlchemy, the code implementation is easy to read and alter.

Finally, the output of the loader is a new analysis record in the database that is associated with the newly generated sample sheet. This new record contains all the information necessary to run the analysis.

4.2.4. The workflow management system

As the fourth component of the infrastructure, we used CloudConductor as the workflow management system. CloudConductor is the central component that receives

all the necessary configuration files from the managing daemon and performs the actual biological analysis of the input samples. The provided configuration files are the sample sheet generated by the data loader and the resource kit, pipeline graph and platform configuration from the database associated to the selected pipeline.

CloudConductor is the only component that does not interact backwards with the other components of the infrastructure. It is launched and supervised by the managing daemon. The input configuration files are obtained from the database, while the output data and the final report are returned by CloudConductor to the preferred storage system defined by the user in the platform configuration file. The final report contains runtime information about each module from the pipeline and paths to all generated files. Thus, in order to understand the results of the analysis run, the managing daemon only needs to parse and understand the final report.

CloudConductor is the platform-agnostic element of the infrastructure. Depending on the platform configuration file, individual instances can be provisioned on different cloud platforms, allowing the system to utilize resources from different sources. CloudConductor will generate additional instances on the specified cloud platform and perform the analysis routines on them, by obtaining the resources available in the resource kit or the input data specified by the sample sheet and parent modules output. When completed, CloudConductor will clean-up all resources allocated by it.

The workflow management system component of the infrastructure was implemented to be modular, thus the user can modify or add a new preferred processing system to the infrastructure.

4.2.5. The managing daemon

The fifth and last component of the infrastructure is the managing daemon. It is the central component in charge of managing and monitoring the execution of pipelines defined in the database by the loader.

The main function of a managing daemon in the infrastructure is to perform two steps. The first step is the launching of new analyses and it involves periodically checking the database for new analysis entries. When a new analysis entry is discovered, the managing daemon creates a new environment for the workflow management system and launches it. The environment preparation is represented by creating a new instance on a preferred cloud platform and provisioning the input configurations on it. Besides launching the analysis in the newly created environment, the managing daemon has to supervise the process.

The second step of the managing daemon is the report parsing. After an analysis is complete, the managing daemon needs to read and interpret the generated report. The generated report is the only method of communication between the workflow management system and the user. All the output information generated by the workflow management system will be present in the final report and the managing daemon has to extract it and persist it in the database.

A managing daemon can be implemented in multiple methods. The simplest way is to implement a main daemon server that has two additional daemons running for each of the above functionality. The two daemons will periodically check for new analysis reports in the database and new generated reports in the storage system, while the main daemon server will ensure that the whole system is alive and functional. A more complex but scalable way is to use an orchestration system, such as Kubernetes, to manage the daemon processes.

In our implementation we chose Kubernetes as the main system to launch CloudConductor on Google Cloud Platform. Kubernetes is an orchestration system that allocates resources on-demand and performs jobs requested by the user. In the Kubernetes system, the daemons are two jobs that are periodically being executed. One job performs the analysis launch, while the second job identifies and parses the final reports generated by CloudConductor. The processing in Kubernetes is done by generating a node on which the processing environment is setup. Next, Kubernetes launches the analysis in the new node using CloudConductor and the configuration files present in the database.

4.3. Example usage

In this section, we will present the entire lifespan of an analysis process through the infrastructure system. We will use the transcriptome pipeline from CloudConductor's chapter (Figure 6) to present how we managed to run it using our system.

The first thing that needs to be ensured prior to starting to process the transcriptome data is that all the necessary information is present in the database. The user can use the website to ensure all the samples are available for analysis.

Next, the user would select all the samples to be analyzed and select a specific pipeline to analyze them. In our case, we implemented a pipeline named “RNA_SEQ” that will analyze transcriptome sequencing data.

Using the website, the user can select the samples and run the preferred pipeline. After this action, the website retrieves each sample’s data, parses it and requests the loader to setup the analysis using the transcriptome sequencing pipeline. Each request is represented as a new record in the infrastructure’s database. Notably, this will be the only manual step required by the user to perform this analysis.

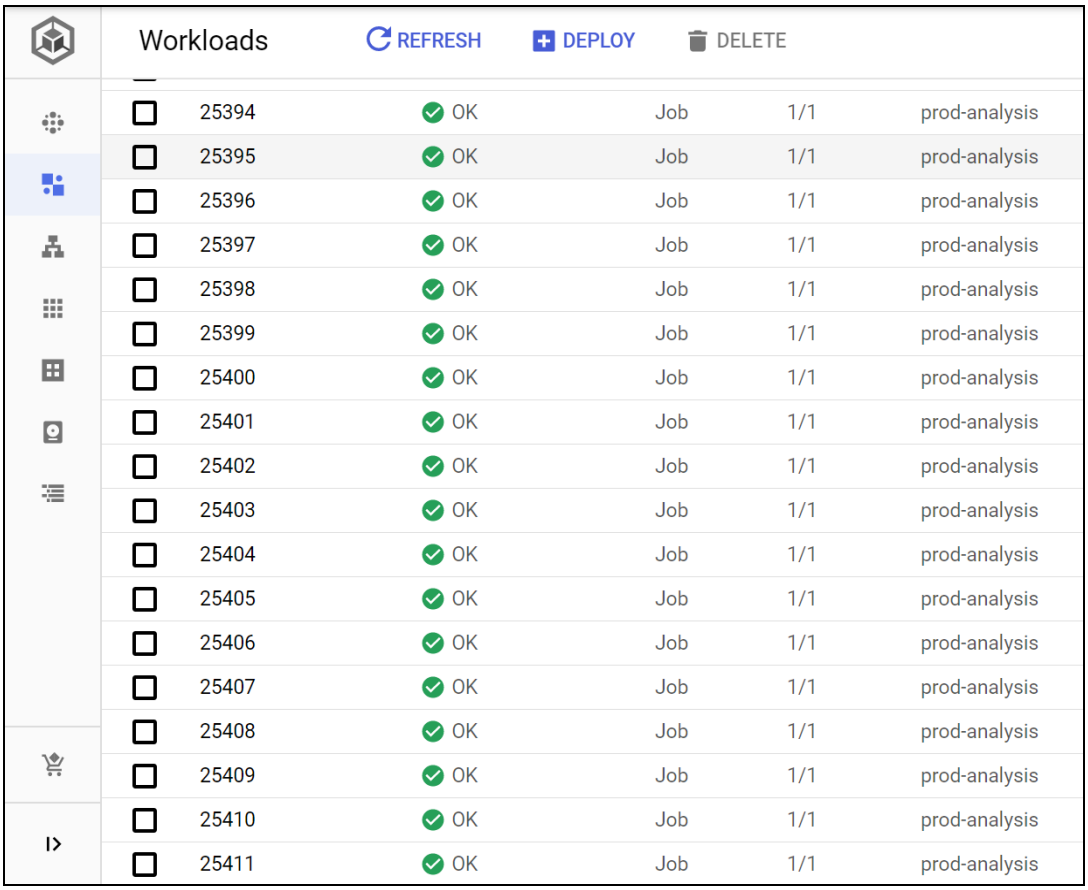


Figure 22. List of analyses launched by Kubernetes. Each record represents a job generated by Kubernetes. For each job, Kubernetes will create a new node and run CloudConductor on it.

After the analysis has been requested, in the background the data loader reads the request metadata and parses the sample and sequencing data associated with it. Each request becomes a new analysis record. Each new analysis record is then associated with the configuration files, including a newly generated sample sheet.

In the meantime, the managing daemon, which is present on Kubernetes, identifies the new analysis record addition in the database and starts processing it. Kubernetes allocates a new node on the Google Cloud, transfers all the necessary

configuration files and starts running the pipeline using CloudConductor. Kubernetes processes multiple analysis records at the same time, thus ensuring the scalability of the entire infrastructure. Figure 22 shows a list of analysis requests that are being processed by Kubernetes.

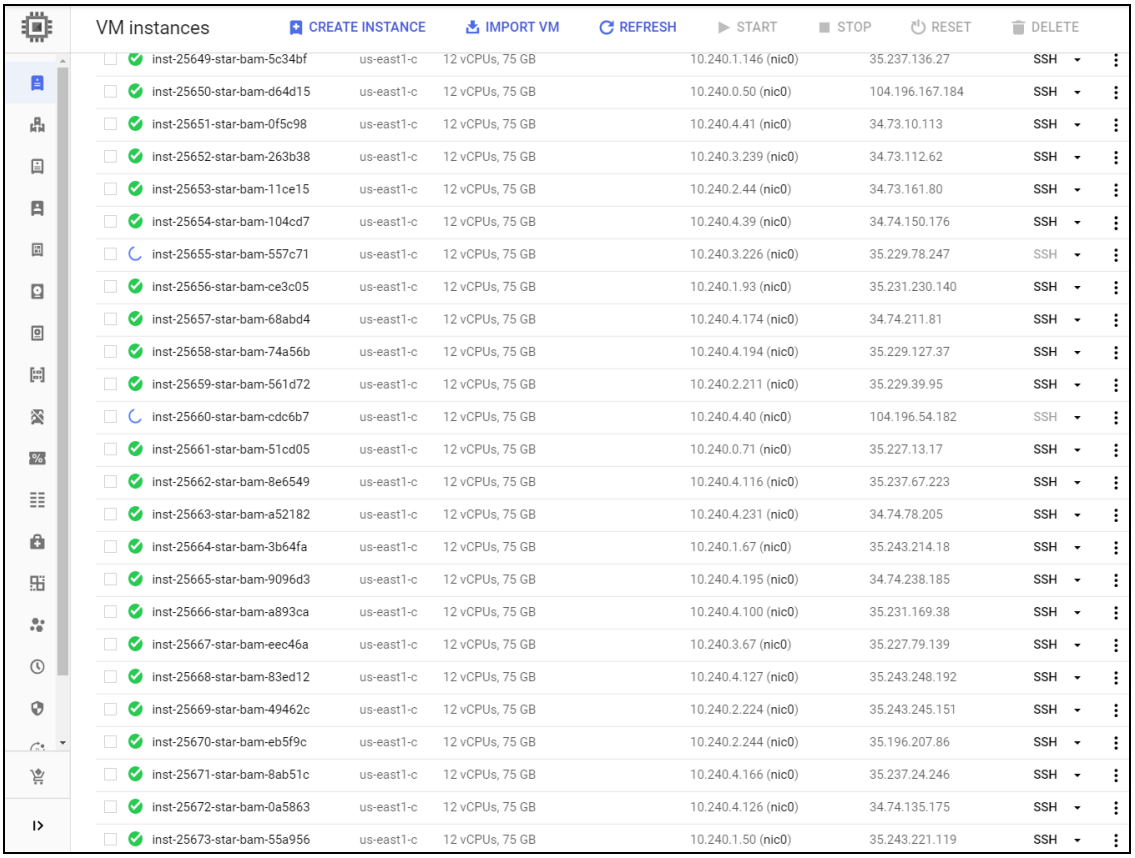


Figure 23. List of instances launched by CloudConductor. Each record represents an instance that is associated with a submodule in the analysis pipeline. As the name of the instances suggest, each instance is running STAR on a different sample.

After the steps done by the managing daemon, each CloudConductor run starts running the pipeline assigned to it. CloudConductor's progression can be observed by the creation of the instances on the cloud platform. Figure 23 shows a part of all the instances

that are generated by CloudConductor for the analysis to take place. In the meantime, the website is presenting the progress on running the 775 samples on the homepage of the user that initiated the analysis. Figure 24 presents the progress bar of the analyses present on the website. Thus, the user can obtain the analysis progress in real-time.

Finally, after all the runs have completed, the user can see the status of all the analyses on the website and CloudConductor jobs' status in the Kubernetes. If any of the analyses failed, the user will be informed about the encountered error to enable debugging. After all analyses ran successfully, the user will be able to get access to the output files that will be used for the downstream analysis. Depending on the implementation design, the analysis results can be summarized on the website as well.

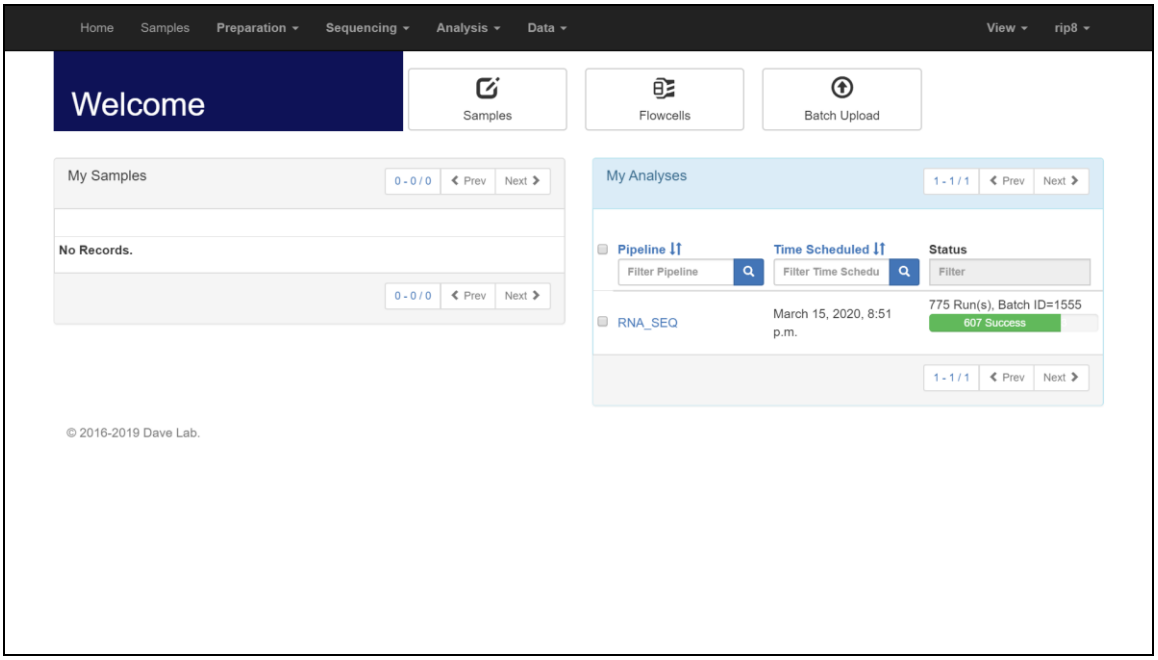


Figure 24. Website interface showing the progress on the transcriptome analysis. The website is presenting in real-time the analysis progress.

CHAPTER 5. Conclusion

As the size of biological data are continuously increasing and additional new methods of analysis are being developed, there is a strong need for computing systems that automate the bioinformatics process. This would certainly improve the feasibility and human costs of processing large cohorts of samples. Using the right tools, a cloud-based infrastructure could help researchers analyze data regardless of institutional infrastructure and resources.

We have developed CloudConductor as an approach to analyze biological data using cloud computing in a scalable and cost-efficient manner. By implementing additional infrastructure components to expand CloudConductor's functionality, multiple samples can now be processed in parallel using an efficient analysis system. Using this automated infrastructure, we have characterized the genetic alterations in a clinically important cancer (Burkitt Lymphoma). Overall, the work in this thesis provides an example of how biomedical research can be optimized and accelerated by utilizing methods that leverage cloud computing resources.

In CloudConductor, there are two future improvements that could be implemented to extend the functionality of the framework. Both improvements are related to the platform-agnostic abstraction. While we have achieved complete abstraction for the compute services of any cloud platform, we have not done so for storage services. A complete abstraction would require automatic authentication and functionality of the services on a cloud platform.

Thus, the first improvement that should be considered in CloudConductor is a complete abstraction of all cloud storage systems. Currently, CloudConductor allows the users to define the commands for general storage operations, such as copying a file or checking its existence, on any cloud platform. This approach is not completely platform agnostic as these commands can differ based on the direction of transfer and each cloud system has a unique way of transferring data and authentication. Consequently, the only way to transfer data in any direction from and to different cloud storage systems is by using a universal protocol of representing data, i.e. the binary format. Storage abstraction can be achieved by representing the data in binary format during transfers, such that each cloud system will read data into binary format and will write binary format into their storage format.

In the new improved implementation of the cloud storage abstraction, an abstract class CloudStorage could be extended for each cloud storage provider. The user will need to implement two methods. The first method is reading the data and streaming it in binary format, while the second method is writing a stream of binary data.

Further, the second improvement to be implemented in CloudConductor would be the second described level of granularity, i.e. platform agnostic abstraction. As described in the CloudConductor chapter, there are two levels of processing granularity in which CloudConductor could be implemented to be platform agnostic. First level of granularity would be at the pipeline level in which one single cloud platform is used to process the entire pipeline, while the second level is at the module level in which each pipeline

module can be processed on a different cloud platform. Implementing the second granularity level would allow the users to process different steps of the analysis on their preferred cloud platform. For example, certain modules would be useful to be processed on the cloud platform on which the input resources are already present, thereby reducing costs that might be incurred by data transfer.

A platform-agnostic implementation at the module-level granularity can be implemented if any running cloud instance is able to transfer data to and from any cloud storage by using the complete storage abstraction described above. Additionally, the pipeline graph implementation will need to be modified to allow the user to specify platform configuration settings for each module.

Our work shows the power of combining these approaches to unravel the genomics of Burkitt lymphoma. We identified that EBV infection is associated with a higher mutational burden in Burkitt lymphoma and additional clues to the viral contribution to the oncogenesis and tumor evolution. Our data identifies new genomic markers and regulatory elements that could enable new therapeutic and clinical approaches that are sorely needed in the resource-poor environments where most BL patients reside. Additionally, our data enable new approaches to make the clinically important distinction between BL and DLBCL, which share many phenotypic similarities, but require very different treatments. Our work highlights the additional work needed to better understand the pathogenesis of Burkitt lymphoma which arises through complex interactions between the host and EBV viral genomes. Our data provide

a starting point for developing clinically relevant models for the disease that represent the complete spectrum of the human disease.

We envision a number of improvements that can be added to the current design and implementation of the automated infrastructure. These include enhancements associated with security, infrastructure adaptability and real-time process logging and debugging.

First, security of the data is a particularly important element when working with patient data. Any bioinformatics infrastructure has to ensure that the user can store sensitive patient data in a manner compliant with HIPPA and other current privacy policies. Currently, the security level of the infrastructure implementation is based on the security offered by each component. By default, cloud platforms offer multiple layers of security, including service authentication and SSH key pairing. Individual components such as the website and the data loader may also require user authentication for accessing data and sending requests. However, an overall layer of security on top of the entire infrastructure would be beneficial by ensuring that, if the developers change or add components that create a security breach in the system, the entire infrastructure does not become compromised.

Second, improving the system adaptability is an important direction in bioinformatics infrastructure development[42]. Currently, not all components of our infrastructure can adapt to the type of data that they are provided. Unlike CloudConductor, which can adapt to the platform and input data provided, infrastructure

components like the data loader can currently only handle genomics data. Implementing an infrastructure that is able to adapt to different types of input would allow researchers from different scientific fields to follow similar standards. Previous work has already been done to standardize the method of communication to workflow management systems using a common workflow language (CWL)[43], but additional work is necessary to standardize the other components of the infrastructure.

Finally, as large cohorts of samples are used in a computing infrastructure, it is important to ensure real-time process logging and debugging. Currently, in the presented infrastructure design, the database would record the analysis progress at specific timepoints of the whole processing and display them on the website to communicate them to the user. For example, in case of a failed analysis run, the failed status and the reason of failure is reported to the user on the website. This requires the user to identify, understand and fix the problem before attempting to retry the analysis. Recent automated bioinformatics infrastructures[14,44] have implemented fault tolerant or "self-healing" features. Using these features, the system analyses the process logs and tries to automatically fix appearing issues. For example, in CloudConductor, we have implemented a fault-tolerant system when using preemptible instances to reduce the total cost of processing. When a process on a preemptible instance has failed, CloudConductor examines the error log and decides if the failure is associated to a preemption event. In case of a preemption event, the whole instance follows a "regenerative" procedure to automatically rerun the process. As a future development, in addition to this

implementation, a new self-debugging component could be added to the infrastructure. This component will either analyze appearing errors and automatically apply corrections or introduce checkpoints in the system that will prevent further encounter of similar errors.

This work represents a critical starting point for our group for the systematic application of large-scale genomics and a scalable approach that greatly improves the efficiencies of both the bioinformatics analyses and the individuals that perform them.

On a personal note, I have started graduate school with two goals in mind. First, I always had a strong interest in biology and I wanted to acquire a better understanding of complex systems such as cancer development or other general processes that underlie human biology. Second, I wanted to channel my passion for computer science by applying diverse computing methods and algorithms to enable a better understanding of complex biology. Although biology has turned out to be far more complex than many of us ever imagined, and the computing challenges have been difficult to solve, my PhD journey and this dissertation have provided fertile ground to pursue both goals productively. These interests will continue to motivate my future work.

Bibliography

1. Hanahan, D. & Weinberg, R. A. Hallmarks of cancer: the next generation. *Cell* 144, 646–674 (2011).

2. Hanahan, D. & Weinberg, R. A. The hallmarks of cancer. *Cell* 100, 57–70 (2000).

3. Hedenfalk, I. *et al.* Gene-Expression Profiles in Hereditary Breast Cancer. *N. Engl. J. Med.* 344, 539–548 (2001).

4. Bray, F. *et al.* Global cancer statistics 2018: GLOBOCAN estimates of incidence and mortality worldwide for 36 cancers in 185 countries. *CA. Cancer J. Clin.* 68, 394–424 (2018).

5. Swerdlow, S. H. *et al.* T. *Blood* 127, 2375–2390 (2016).

6. Sanger, F., Nicklen, S. & Coulson, A. R. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. U. S. A.* 74, 5463–5467 (1977).

7. Lander, E. S. *et al.* Initial sequencing and analysis of the human genome. *Nature* 409, 860–921 (2001).

8. Illumina, I. An introduction to Next-Generation Sequencing Technology. (2017).

9. Shalem, O. *et al.* Genome-scale CRISPR-Cas9 knockout screening in human cells. *Science* 343, 84–87 (2014).

10. Muir, P. *et al.* The real cost of sequencing: scaling computation to keep pace with data generation. *Genome Biol.* 17, 53 (2016).

11. Campbell, P. J. *et al.* Pan-cancer analysis of whole genomes. *Nature* 578, 82–93 (2020).

12. Vivian, J. *et al.* Toil enables reproducible, open source, big biomedical data analyses. *Nat. Biotechnol.* 35, 314–316 (2017).

13. Di Tommaso, P. *et al.* Nextflow enables reproducible computational workflows. *Nat. Biotechnol.* 35, 316–319 (2017).

14. Yakneen, S., Waszak, S. M., Gertz, M. & Korbel, J. O. Butler enables rapid cloud-based analysis of thousands of human genomes. *Nat. Biotechnol.* 38, 288–292 (2020).

15. Leipzig, J. A review of bioinformatic pipeline frameworks. *Brief. Bioinform.* 18, 530–536 (2016).

16. Wercelens, P. *et al.* Bioinformatics Workflows With NoSQL Database in Cloud Computing. *Evol. Bioinforma.* 15, 1176934319889974 (2019).

17. Panea, R. I. *et al.* The whole-genome landscape of Burkitt lymphoma subtypes. *Blood* 134, 1598–1607 (2019).

18. Reddy, A. *et al.* Genetic and Functional Drivers of Diffuse Large B Cell Lymphoma. *Cell* 171, 481-494.e15 (2017).

19. Love, C. *et al.* The genetic landscape of mutations in Burkitt lymphoma. *Nat. Genet.* 44, 1321–5 (2012).

20. Richter, J. *et al.* Recurrent mutation of the ID3 gene in Burkitt lymphoma identified by integrated genome, exome and transcriptome sequencing. *Nat. Genet.* 44, 1316–1320 (2012).

21. Schmitz, R. *et al.* Genetics and Pathogenesis of Diffuse Large B-Cell Lymphoma. *N. Engl. J. Med.* 378, 1396–1407 (2018).

22. Kaymaz, Y. *et al.* Comprehensive Transcriptome and Mutational Profiling of Endemic Burkitt Lymphoma Reveals EBV Type-Specific Differences. *Mol. Cancer Res.* 15, 563–576 (2017).

23. Bolger, A. M., Lohse, M. & Usadel, B. Trimmomatic: A flexible trimmer for Illumina sequence data. *Bioinformatics* 30, 2114–2120 (2014).

24. Li, H. & Durbin, R. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics* 26, 589–595 (2010).

25. Cibulskis, K. *et al.* Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nat. Biotechnol.* 31, 213–219 (2013).

26. Wang, K., Li, M. & Hakonarson, H. ANNOVAR: Functional annotation of genetic variants from high-throughput sequencing data. *Nucleic Acids Res.* 38, 1–7 (2010).

27. Dobin, A. *et al.* STAR: ultrafast universal RNA-seq aligner. *Bioinformatics* 29, 15–21 (2013).

28. Li, B. & Dewey, C. N. RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome. *BMC Bioinformatics* 12, 323 (2011).

29. Subramanian, A. *et al.* Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proc. Natl. Acad. Sci. U. S. A.* 102, 15545–15550 (2005).

30. Rausch, T. *et al.* DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics* 28, i333–i339 (2012).

31. Buchfink, B., Xie, C. & Huson, D. H. Fast and sensitive protein alignment using DIAMOND. *Nat. Methods* 12, 59–60 (2015).

32. Lawrence, M. S. *et al.* Mutational heterogeneity in cancer and the search for new cancer-associated genes. *Nature* 499, 214–218 (2013).

33. Gu, Z., Gu, L., Eils, R., Schlesner, M. & Brors, B. circlize Implements and enhances circular visualization in R. *Bioinformatics* 30, 2811–2812 (2014).

34. Ester, M., Kriegel, H.-P., Sander, J. & Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. in 226–231 (AAAI Press, 1996).

35. Ou, J. & Zhu, L. J. trackViewer: a Bioconductor package for interactive and integrative visualization of multi-omics data. *Nature methods* vol. 16 453–454 (2019).

36. Jiang, Y., Soong, T. D., Wang, L., Melnick, A. M. & Elemento, O. Genome-wide detection of genes targeted by non-Ig somatic hypermutation in lymphoma. *PLoS One* 7, e40332 (2012).

37. Arthur, S. E. *et al.* Genome-wide discovery of somatic regulatory variants in diffuse large B-cell lymphoma. *Nat. Commun.* 9, 4001 (2018).

38. Zhang, J. *et al.* The genomic landscape of mantle cell lymphoma is related to the epigenetically determined chromatin state of normal B cells. *Blood* 123, 2988–2996 (2014).

39. Alexandrov, L. B. *et al.* Signatures of mutational processes in human cancer. *Nature* 500, 415–21 (2013).

40. Yu, G. & He, Q.-Y. ReactomePA: an R/Bioconductor package for reactome pathway analysis and visualization. *Mol. Biosyst.* 12, 477–479 (2016).

41. Wang, T. *et al.* Identification and characterization of essential genes in the human genome. *Science* 350, 1096–1101 (2015).

42. Boyle, J. *et al.* Adaptable data management for systems biology investigations. *BMC Bioinformatics* 10, 79 (2009).

43. Amstutz, Peter; Crusoe, Michael R.; Tijanić, Nebojša; Chapman, Brad; Chilton, John; Heuer, M. Common Workflow Language, v1.0. (2016).

44. Dean, J. & Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 107–113 (2008).

Biography

Razvan Ioan Panea was born on August 1992 in Romania. He pursued his undergraduate studies at Jacobs University Bremen in Germany. On May 2014, Razvan received a Bachelor of Science degree in Applied and Computational Mathematics with specialization in Computational Biology. During his time in Bremen, he worked as an intern, and afterward as a software engineer, for the SILVA project at Max Planck Institute for Marine Microbiology.

After his undergraduate studies, Razvan joined Duke University to continue his studies as a graduate student in Sandeep Dave's lab. On May 2020, Razvan obtained his Ph.D. degree in Computational Biology and Bioinformatics.