

Modern systems for large-scale genomics data processing in the cloud

Sergei Yakneen
EMBL
Meyerhofstrasowse 1,
69117 Heidelberg, Germany

09/04/2019

Contents

1	Introduction	6
1.1	Context and Motivation	6
1.2	Challenges and Problem Statement	7
1.3	Proposed Solution	9
1.4	Thesis Outline	13
2	Background and Related Work	14
2.1	Genomics	14
2.1.1	History of Genomics	14
2.1.2	Next Generation Sequencing	16
2.1.3	Genomics Studies	17
2.1.4	Cancer Genomics	17
2.2	Computational Methods for Next Generation Sequencing	18
2.2.1	File Formats	24
2.2.2	Alignment	35
2.2.3	Raw Data QC	51
2.2.4	Germline SNP Calling	56
2.2.5	Germline Structural Variant Calling	68
2.3	High Performance , High Throughput, and Cloud Computing	75
2.4	Workflow Systems	78

3 The Butler Framework - Requirements and Architecture	81
3.1 Functional Requirements	81
3.1.1 Access to Data	82
3.1.2 Access to Compute Capacity	84
3.1.3 Implementations of Scientific Algorithms	86
3.1.4 Workflow Definitions	88
3.1.5 Workflow Engine	91
3.1.6 System of Record	94
3.1.7 Troubleshooting Errors	94
3.2 Non-functional Requirements	101
3.2.1 Scalability	101
3.2.2 Availability	103
3.2.3 Ease-of-use	105
3.2.4 Interoperability	105
3.3 General Design Principles	106
3.3.1 Existing Open-Source Software	106
3.3.2 Service Orientation	107
3.3.3 Cloud Agnostic	108
3.3.4 Open Source License	110
3.3.5 Overall System Design	110
3.4 Cluster Lifecycle Management	110
3.4.1 Terraform	111
3.4.2 Terraform Use in Butler	114
3.5 Cluster Configuration Management	115
3.5.1 Saltstack	116
3.5.2 Saltstack Use in Butler	121
3.6 Workflow System	125

3.6.1	Workflow System Overview	125
3.6.2	Workflow Definition	126
3.6.3	Analysis Tracker	128
3.6.4	Workflow Configuration	135
3.6.5	Workflow Runtime Management	137
3.7	Operational Management	139
3.7.1	Monitoring Metric Collection	140
3.7.2	Monitoring Visualization	142
3.7.3	Server Log Collection and Visualization	145
3.7.4	Self-Healing	148
3.7.5	Service Discovery	156
4	The Butler Framework - Implementation and Experimental Validation	158
4.1	Deployment on EMBL/EBI Embassy Cloud	158
4.2	PCAWG Germline Analyses	162
4.2.1	Variant Discovery	163
4.2.2	Variant Genotyping	168
4.2.3	Variant Filtration	169
4.2.4	Genotype Phasing	169
4.2.5	Data Submission	170
4.2.6	Structural Variant Calling	170
4.3	Experimental Runs	171
4.3.1	Freebayes Common Variant Genotyping	171
4.3.2	Freebayes Variant Genotyping	173
4.3.3	Delly Full Variant Genotyping	176
4.4	Performance Evaluation and Comparison	180
4.5	System Design Recap	184

5 The Rheos Framework	186
5.1 General Framework Design	186
5.2 Data Streaming Architecture	188
5.2.1 Service-Oriented Data Streaming Model	189
5.3 Domain-specific Problems	195
5.3.1 Read QC Metrics	197
5.3.2 Alignment	201
5.3.3 Simple Germline SNP Calling	211
5.3.4 Germline Structural Variant Calling	218
5.4 Rheos implementation	225
5.4.1 Rheos Service	226
5.4.2 Queueing	229
5.4.3 Partitioning	230
5.4.4 Deployment	233
5.4.5 Services of Rheos	237
5.5 Experimental Validation	243
5.5.1 Deployment	243
5.5.2 Sample Selection and Preparation	245
5.5.3 Germline SNP Calling	246
5.5.4 Germline Deletion Calling	252
6 Discussion and Conclusion	258
6.1 Validation and Conclusion	258
6.2 Future Direction	259
Appendix	260
A Code Listings	260

Chapter 1

Introduction

1.1 Context and Motivation

In the 17 years since the publication of the first draft human genome[111] the fields of genomics and molecular biology have undergone a major shift. The direction of this shift is towards an increasing adoption of computational approaches alongside experimental methods, bringing both of these fields of study into the realm of information science. This transition has been facilitated by two major factors - the advent of next generation sequencing[192], and the development of the Internet and cloud computing[28]. Next generation sequencing has been responsible for bringing down the cost of DNA sequencing to the point where it has become possible to sequence and study entire populations of individuals[86], while the Internet and cloud computing are democratising access to large-scale computational resources such that computation on big datasets, which was previously only accessible to large institutions, is becoming tractable to a growing group of researchers and citizen scientists.

The continued appetite for sequencing of larger and larger cohorts of individuals by the research community is driven by the desire to better understand the evolutionary history of the human species[102], to identify causes and mechanisms of action of rare genetic diseases that affect a very small proportion of the population[25], and to elucidate and potentially target the genetic component of more common diseases such as cancer[227], heart disease[26], or dementia[194] that place a heavy burden on our society. All of these factors together mean that the need for the generation and interpretation of genomic data is growing at an unprecedented scale.

Yet, the analysis of DNA sequencing data to study human genomes remains a largely unsolved problem. The protein coding sequence of the human genome, its *exome*, constitutes roughly 1% of the human DNA and successful studies have carried out exome-based analyses on cohorts at the scale of tens of thousands of individuals[119]. However, the other 99% of the human genome, its non-coding regions, contain crucial information such as gene regulatory elements[39] that are essential to our full understanding of the mechanisms and processes that are underlying the human genetic landscape. Given current technologies, Whole Genome Sequencing

(WGS) is considerably more expensive and generates data-set sizes at the petabyte (PB) scale that are challenging for even the largest international consortia to tackle [204]. WGS studies at 100,000 participants scale that are planned for the coming years[59] will further increase data-set size and complexity by several orders of magnitude, a challenge that is presently unanswered by the current generation of bioinformatics infrastructures and algorithms.

A bigger and more distant challenge is the development of clinical sequencing and genomics which will truly bring whole-genome sequencing applications to population scale. Currently DNA sequencing has limited adoption within the clinical practice with applications limited to rare Mendelian disorders[118] and certain types of cancers[184] where a small set of genomic loci is interrogated via a gene panel[15] with a set of well-delineated disease sub-types based on these genetic markers. The use of whole-genome sequencing for clinical applications is presently nearly non-existent due to its high cost compared to the clinical utility of its findings, yet the potential for the impact of this approach remains substantial as certain genomic variants such as Structural Variations (SVs) typically have a large effect on an individual's phenotype due to their size[174], but are generally not amenable to interrogation via gene panels.

The magnitude of the opportunity for improvement in the space of DNA sequencing and genomics is thus clear to us - we seek a way to improve the current methods of DNA data analysis such that it becomes tractable and cost-effective to undertake whole-genome sequencing studies within research and clinical contexts at the scale of hundreds of thousands to millions of human genomes.

1.2 Challenges and Problem Statement

Let's examine the key challenges that need to be addressed in order to enable efficient genomic data analysis at the scale that is desired by the research and clinical communities.

Several broad groups of challenges are identified below and further examined throughout this thesis:

Data Set Size - The size of the raw genomic data generated by population-scale studies will be hundreds to thousands of petabytes making it impractical to move and make copies of the data[205].

Data Retention - The cost of generating the data is significantly higher than the cost of storing the data, thus making it impractical to throw away the raw data after initial analysis[156].

Data Formats - The data formats used for storing genomic data are primarily large size character and binary files (FASTA, SAM, BAM, VCF)[135, 49] that have loose specifications and scale poorly to large cohort sizes. File indexing

structures typically support indexing by genomic coordinate only, thus limiting queryability.

Data Fragmentation - The data will be generated at multiple sequencing centres located in different jurisdictions with a wide variety of genomic data handling requirements. Data processing must proceed at multiple locations that respect the requirements of each jurisdiction[154].

Data Type Diversity - Comprehensive characterization of a person's genome that is useful in a clinical setting implies the collection and integrative analysis of many diverse data types - including germline[140] and somatic[82] genomic variants, transcriptomics[224], epigenomics[103], metabolomics[220], and clinical information. Uniform collection, processing, and integration of these data types is required to successfully associate the role of this genomic variation on disease phenotypes[184].

Data Processing Stages - Data processing for genomics analysis proceeds through a sequence of stages from base-calling, to quality-control, to genome alignment, to variant calling, to annotation, to downstream analysis[54]. Each stage typically has non-trivial computational requirements needing several days on a multi-core machine to complete with increased failure risk as a function of data set size. Intermediate results from one stage are often required as input for downstream stages. Fully sequential processing makes inefficient use of the data by redundantly loading and interrogating the data in memory over a series of passes through the sample.

Toolset Fragmentation - Although comprehensive genomic characterisation of each sample is typically of interest to researchers, specific bioinformatics tools only provide solutions to a limited subspace of the overall problem, thus requiring integration of multiple tools that may produce incongruent outputs and compete for resources producing computational bottlenecks.

Having listed these challenges we attempt to restate the problem in simpler terms before providing a high level overview of the types of approaches and solutions that will be developed and considered in detail in the body of this thesis in order to deliver a conceptual and practical framework for the effective management of genomic data at the desired scale.

Our problem statement is then as follows:

Human genomic data sets will, in the future, be generated for analysis in various locations throughout the world, at the aggregate rate of multiple petabytes of data per day in the context of disease and clinical practice. The desired outcome of these analyses is the comprehensive characterisation of genomic features and their association with phenotypic variables of interest[230]. The goal of the research community is in capturing the maximum number of samples – N , with high accuracy – A , to increase statistical power of studies[97], while the interest of clinicians is to capture specific individuals with high accuracy – A , and in the shortest possible time – T , in order to inform clinical decision making[221]. Both parties wish to do so at

minimal possible total cost – $C = c_g + c_s + c_a + c_r$, taking into account the cost of data generation, cost of data storage, cost of data analysis, and cost of subsequent data retrieval. Because of the high cost of generating this data each time, the data, once generated, will need to be stored for the foreseeable future. The overwhelming data set size prevents data movement between locations, requiring analysis algorithms to be colocated with the data.

The analysis is hampered by reliance on data formats that have not been designed for operation at such large scale and the necessity to execute a variety of computational algorithms[131, 242, 14, 150] on the data that have been individually developed by different authors within an academic context, using different technologies that compete with each other for computational resources, and at-times produce contradictory results that require human intervention to integrate. The underlying assumption of genomic coordinate-sorted ordering and traversal of the data made by most algorithms limits the modes of reasoning about the dataset to a series of pre-processing steps, followed by another series of coordinate-wise traversals through the data, which impose severe processing time costs, such as the requirement to have generated, seen and sorted all of the data, before an analysis can proceed as well as the inability to stop and interpret analysis results mid-processing.

The optimization problem of maximising N , and A , while minimizing C for research purposes remains unsolved for values of N above 3000 samples when it comes to high-coverage whole genome sequencing, while the problem of maximizing A , and minimising T , and C is presently not solved in the clinical setting for any sample size. It is our proposed solution for tackling these issues that we turn to next.

1.3 Proposed Solution

We assume that N , the number of samples that can be successfully sequenced will depend almost entirely on the total cost C , which itself, among other factors, is determined by the desired accuracy and processing time. We thus focus most of our efforts on the joint optimization of cost, accuracy, and time as necessary conditions for the maximisation of effective sample size N and enablement of whole genome sequencing for clinical practice.

We note that the cost of data generation C is dependent on the sequencing technology used, the underlying chemistry, and the cost of the reagents[153]. Improving these characteristics falls outside the scope of our discussion, and we assume the cost of the data generation component c_g of C to be constant throughout this thesis.

We establish and discuss at length the characteristics of the optimization function in the body of the thesis but here note briefly that analysis accuracy A is evaluated along the usual dimensions of sensitivity and specificity and can generally be improved by generating more data for a given sample up to a theoretical maximum inherent in the sequencing technology used and the nature of the analysis algorithms employed. Generation of more data naturally leads to increased analysis time T and cost C . The time to accomplish the analysis can be reduced by

either giving up accuracy (by looking at less data, or using faster but less accurate algorithms[132]), by increasing the level of parallelisation within the computational pipeline i.e. parallelising steps that are currently sequential[112], or by utilising additional computational resources, thereby increasing costs. The various components of cost, in turn, can be optimized by improved data storage and retrieval structures[167] (via multi-level caches and hybrid storage media, for example), by improved-efficiency analysis algorithms, and by reduction of analysis accuracy and increase of analysis time (via cheaper hardware).

It is clear from the discussion above that cost, accuracy, and processing time are not orthogonal concerns i.e. changes in one may lead to changes in the other two. It thus appears that no optimization effort is likely to simultaneously satisfy the requirements of all parties that are interested in large scale genomic analysis, and a successful computational framework for delivering such analyses must allow efficient and dynamic optimization of these parameters to fit the needs of the end user. This is typically not the case for present day genomics frameworks because of the sequential way they look at data[149, 216] i.e. all of the data is generated before it is processed by downstream tools, and accuracy and processing time need to be decided on before launching a set of tools because they step through the genome in coordinate-wise manner.

To address these challenges we develop and describe within this thesis a new computational framework, called Rheos, that is based on the concepts of data streaming, cloud computing, and service orientation to provide a comprehensive toolset for genomic data analysis that can potentially scale to processing of millions of genomes while arming its users with the capability to make timely, responsive, and principled decisions about the tradeoffs between analysis cost, accuracy, and duration.

Three distinct characteristics set Rheos apart from current generation genomic analysis frameworks and each of these allows us tackle some of the issues and challenges described in Section 4.1. These are:

- Service Orientation
- Event and Data Streaming
- Random Data Ordering

Service orientation[60] allows us to decompose the overall problem of comprehensively reasoning about genomic data into a set of small loosely-coupled components, each of which is optimized to tackle a particular well-defined subset of the complete set of requirements of the system. Each service has a contract that it makes with its clients, it has an explicit set of inputs that it knows how to process, it has an interface that defines the modes of communication it supports, it has a set of outputs that it produces according to its capabilities, and it has a set of operational characteristics that makes explicit commitments about the service's reliability, speed, etc[168]. This has a number of benefits - a service can be small enough that it optimizes the solution to a particular problem without being subject to the same

competing constraints that larger tools are subject to, which provides opportunities for improved performance and hardware utilization. As long as the service respects its input and output commitments it is free to maintain arbitrary internal representations of the data enabling optimization of data storage and query costs (c_s and c_r). A service can be monitored such that hardware is allocated elastically up and down based on demand to ensure optimal utilization, as well as providing a continued measure of whether the service is meeting its operational reliability requirements to its clients[43]. This is especially useful in contexts where demand for certain calculations is highly variable.

The issue of inter-service communication is of major importance because of the large size of the data-set and the potential for various difficult-to-debug run-time race and error conditions inherent in a distributed system[76]. Currently, most bioinformatics tools do not communicate with each other directly via an API, instead they use popular file formats such as SAM/BAM/CRAM[135, 74], and VCF[49], as well as a myriad of more esoteric file formats not only as a storage medium but also as a means of communicating information between each other. This paradigm hurts the ultimate scalability of the entire system because of the necessity to write data to disk and possibly move it over the network in order to enable communication across tools. Furthermore, a file-based information exchange mechanism forces a coarse-grained, sample-level, communication between components that wish to avoid tight coupling between each other, even though most of the reasoning about genomic data occurs at locus, or small locus-neighbourhood, levels[57].

Rheos adopts a data and event stream approach to accomplish scalable fine-grained communication between services[157]. This approach allows each service to listen to and produce data at the level of granularity that it needs to make decisions, and that its downstream dependencies are interested in (for instance at read, locus, or breakpoint levels). When primary data is ingested into the Rheos system (from a sequencer, or a data repository) the data stream can start to be analyzed immediately[87], unlike file-based systems that need to wait for the entire sample to transmit before beginning. This approach can potentially enable real-time analysis given sufficient allocation of computational resources[11]. Since the raw data is extremely large, it is advantageous to move this data between machines, and between disk and RAM as little as possible, thus instead of passing the raw data around the network various services pass around events of interest about the raw data amongst themselves[61]. When a particular service needs the raw data (rather than the corresponding events) for its decision-making it can be shipped this data as necessary, or it can be instructed to run on the host that has already cached this data in memory. Data streaming allows for extreme scalability, but a key challenge when dealing with data streams is that one is no longer guaranteed to ever be able to see "all of the data" for a particular sample, at least in any meaningful amount of time[75]. Because genomic algorithms frequently make use of various summary statistics accumulated over the data-set[170, 122], not being able reason over all the data at once means that approximations for these summary statistics are required. Rheos uses approximations calculated within time windows over the data stream[51, 18] and we consider their properties in detail in the body of this thesis.

A key assumption made by nearly all algorithms in the genomics space that

participate in variant calling and reason over sequence reads is that the reads are coordinate-sorted with respect to the reference genome to which they are aligned[135, 78, 32, 180]. The algorithms then proceed by traversing the genome in coordinate-wise fashion from the beginning of chromosome 1 to the end of chromosome Y interrogating each locus in turn by examining the set of reads that overlap that locus (a read pileup)[135]. Getting the reads into a state that is usable by these algorithms then requires, at a minimum, that all the reads for a given sample have been generated, have gone through QC[231], have been aligned[132], have been investigated for PCR duplicates[216], and have been sorted[216]. Each of these steps can take hours or even days to complete, especially on high coverage whole genome samples. We take a different approach with Rheos by relaxing the requirement for the reads to have been sorted before any variant calling can take place, and instead develop a set of variant calling algorithms that do not assume any particular order within the data that they observe. This allows Rheos to make use of sequence data as soon as it comes off the sequencing machine, thereby dramatically reducing the total time T required to process genomic data compared to the current generation of algorithms. Rheos accomplishes this by employing the service- and stream-based approaches discussed above to process each read on-the-fly as it moves through the system. The read is first assessed for quality, then aligned to a reference genome by the alignment service. This service emits an event with a coordinate that corresponds to the alignment. Variant calling services listen to this event stream and incorporate the evidence for genomic variation supplied by this read into their models of the genomic features that exist at that particular locus for that sample via a statistical framework based on an iterated application of Bayes' rule[239, 21].

Because current generation tools can see all of the data for a particular locus at once they can incorporate all of the evidence supplied by this data in a minimal number of calculations, corresponding to each particular algorithm[135, 78]. Rheos, on the other hand, to incorporate the same amount of evidence will need to perform a larger number of calculations in a redundant manner, incorporating the data as it is observed. This cost is compensated for, however, by the fact that Rheos can immediately incorporate new data about a particular locus when it becomes available without the need to have accumulated all of the data for all of the loci, generating significant time savings. Furthermore, because data arrives in no particular order the set of variant calls produced by Rheos at any given point in time represent a comprehensive characterisation of the sample as if the sample was sequenced at an average coverage consistent with the amount of data that has been observed so far. Observing more data is equivalent to raising the average coverage uniformly throughout the genome, thereby improving call accuracy[13]. This provides us with a framework to actively and dynamically trade off call-set accuracy A for processing time T and cost C as actual data is being observed thereby enabling novel applications whereby sequencing is abandoned early when issues such as sample-swap[64], or contamination[31] are detected. In addition, when sufficient accuracy is reached based on observed data at a particular locus, the framework may choose to stop looking at further data, whereas current generation approaches necessitate committing to a particular sequencing depth a-priori. Furthermore, because current methods iterate through the data in a coordinate-wise manner, their partial results are not really usable until the entire data-set has been traversed (as they represent only

a particular region of the genome), whereas Rheos call-sets represent progressive elaboration of a complete genomic characterisation and are thus usable at any level of accuracy that is fit for the purposes of the underlying analysis. We develop the details of the statistical framework used by Rheos and compare its theoretical and real performance to current generation frameworks in the body of this thesis.

We conceived of Rheos as a modern bioinformatics framework that aims to enable the large scale genomics studies of the future[59, 104, 145] in both research and clinical contexts by providing a toolset that allows for interpretation and comprehensive characterisation of high coverage human whole genome samples at the scale of millions of samples. In order to meet the diverse requirements of its users the framework allows users to make informed and dynamic tradeoffs along the optimization dimensions of cost, accuracy, and time. Rheos unique abilities rest upon three characteristics that set it apart from current generation tools, these are: service orientation, data streaming, and random data ordering. Taken together these characteristics enable Rheos to perform at unprecedented levels of scale while retaining call-set accuracy and reducing per-sample processing time. We dedicate the main body of this thesis to the development of the theoretical framework underlying Rheos, exploring its characteristics, benefits and tradeoffs, discussing its implementation, and evaluating and comparing Rheos' performance to the current best practices in genomics algorithms on real data.

1.4 Thesis Outline

This thesis focuses on the development of the conceptual framework behind the Rheos platform, the theoretical properties of the various algorithms employed by Rheos, and the implementation and experimental validation of the framework on real data. Chapter 2 provides an introduction to the fields of genomics, including cancer genomics and clinical genomics, a survey of the main tools and algorithms that are commonly used in genomics is provided including the details of the underlying statistical models and operational characteristics. The chapter concludes with a look at workflow frameworks that tie individual algorithms together into computational pipelines. Chapter 5 sets up and describes the conceptual framework underlying Rheos based on the approaches of Service Orientation, Data Streaming, and Random Data Ordering, mentioned above. We describe the overall architecture as well as the model behind individual services that comprise Rheos and investigate the theoretical properties of the algorithms that underlie Rheos-based genomic analysis. Chapter ?? describes the actual implementation of the Rheos framework's components and investigates their operational characteristics. Chapter ?? is dedicated to the experimental evaluation of Rheos in comparison to other extant frameworks and algorithms using real genomic data. We conclude this work in Chapter 6 with a discussion of the results and an examination of the future direction of Rheos development.

Chapter 2

Background and Related Work

2.1 Genomics

The field of genomics is closely related to, yet distinct, from the field of genetics, which itself stems from the work of such seminal figures as Charles Darwin[50] and Gregor Mendel[84]. While genetics largely focuses on the study of single (or relatively small numbers of) genes - the *genotype*, and how genetic variation and mutation affect the physical traits of a given cell or organism - its *phenotype*, genomics focuses on larger scale events and mechanisms that tend to act on the entirety of an organism's genome, shaping its architecture and ultimately affecting its survival.

2.1.1 History of Genomics

Each living cell is a bio-chemical machine that carries out a number of complex behaviours such as interactions with the surrounding environment, motility, metabolism, and reproduction, that are necessary for its survival and proliferation, based on a genetic program that is encoded within the cell's DNA. The DNA is nominally subdivided into functionally distinct areas known as *genes*. The cell utilizes the program within each gene by first *transcribing* the DNA into an intermediary information-carrier molecule called RNA, and then *translating* this RNA into molecules called *proteins* that are utilized by the cell to carry out the majority of its functions. Understanding and interpretation of the underlying genetic program thus underpins our ability to comprehend the entirety of the different behaviours that each cell undertakes.

The success of this undertaking is contingent, first and foremost, on our ability to effectively read off the information encoded in the DNA, an activity known as *sequencing*. We are able to sequence DNA thanks to the pioneering work of researchers Rosalind Franklin[73], James Watson, and Francis Crick[225] who first elucidated the physical structure of DNA, then followed by the work of Fred Sanger[189, 188] who devised the first effective DNA sequencing method. The sequencing method

allows us to transform information that is physically encoded on the DNA molecule via a sequence of four distinct types of *basepairs* - Adenine, Cytosine, Guanine, and Thymine into a string stored on a computer using a four-letter alphabet - A,C,T, and G, thus turning DNA interpretation into a digital information processing problem.

While the entire length of the DNA of an organism ranges from several hundred thousand basepairs for simple organisms like viruses and bacteria, to about 3,000,000,000 basepairs for a human, to over 150,000,000,000 for certain plants[171] the limitations of Sanger DNA sequencing technology are such that the sequencing machine can only produce DNA fragment strings, known as *reads* that are 800 - 1,000 basepairs long[189]. Reconstituting the original complete DNA sequence from partial overlaps between reads is thus a costly, time consuming, and computationally intensive problem known as *de-novo assembly*[241]. Once one such full sequence (known as a *reference* sequence) is assembled however, sequencing other individuals of the same (or closely related) species becomes a significantly easier undertaking. Rather than assembling the sequence *de-novo* one can search for a position on the reference sequence that provides the best matching *alignment* between the reference and each read obtained for the specimen under study. This technique is known as *genomic alignment*[131] or *mapping* and yields for each fragment a coordinate that represents where on the reference sequence the fragment maps to. Furthermore, because the DNA of any two organisms of the same species is largely identical, with differences occurring at about 0.1% of all sites (although this depends on DNA mutation rate)[159] researchers are able to significantly reduce the amount of information that is required to fully represent the genome of a specimen by retaining only the information that describes the sites where that specimen is different from the reference sequence for that species.

A general approach has thus emerged, where each new species of interest undergoes a relatively costly *de-novo* assembly process for the first genome, which then becomes the reference genome for that species. The sequencing of further individuals of that species utilizes, relatively cheaper, *alignment* and identification of *variants* (sites where the individual differs from the reference) to investigate the effect these variants may have on different phenotypes of interest such as disease susceptibility and survival[142].

Although genomicists study many different types of organisms the study of human genomes garners by far the most attention and research funding[**needcitation**] due to the natural desire of humans to better understand ourselves and influence, where possible, genetic factors impacting human longevity and health. Subsequent to the development of DNA sequencing methods by Fred Sanger one of the most audacious and crucial projects for the development of genomics as a branch of science has been The Human Genome Project[111] - an international effort to sequence and *de-novo* assemble the first complete human genome consisting of chromosomes 1-22, X, and Y (as well as mitochondrial DNA) and totalling approximately 3 billion basepairs. The project ran for over 10 years, completing in 2001, and cost more than \$3 billion USD. Although the main project effort was completed using the Sanger sequencing method, a competing version of the human genome was simultaneously published by a commercial company led by JC Venter[218], using a new sequencing method called shotgun sequencing[219], a method that formed the basis for a new revolution

in sequencing technology, now termed Next Generation Sequencing[192].

2.1.2 Next Generation Sequencing

The Next Generation Sequencing methodology[144] relies on fragmenting the DNA of a subject into millions of fragments that are between 100-500 basepairs (bp) in length, then sequencing all of the short fragments and aligning all the reads to the reference with the aid of a relatively fast algorithm[132]. Because NGS sequencing methods are prone to certain errors and biases[56], it is necessary to sequence enough DNA fragments to overlap (or cover) every location in the genome several times (typically 10-30), in order to build a statistical model that will be able to determine the underlying sequence, known as *genotyping*[160], with a high degree of confidence. Thus, at present, a single sequenced DNA sample will typically contain 1 billion reads with a file size of 150GB when compressed.

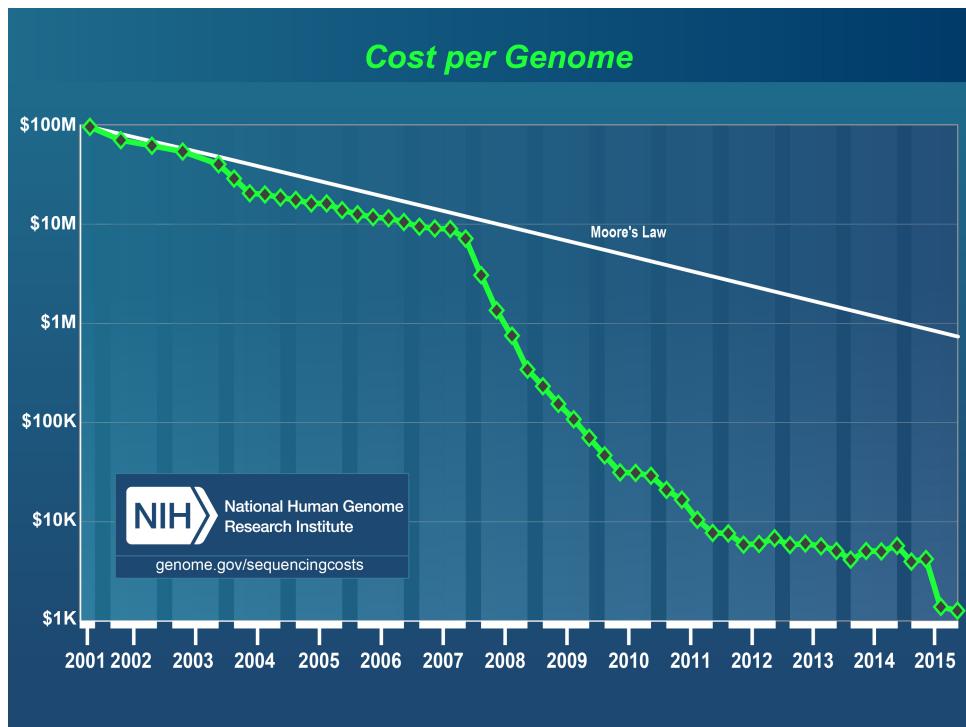


Figure 2.1: Cost of DNA sequencing[212]

Figure 2.1 shows the change in the cost of DNA sequencing over the course of the past 15 years. The precipitous drop in sequencing cost observed since 2008 coincides with wide adoption of NGS methodologies. This drop in price has made tractable a new set of large scale genomics sequencing projects that aim to characterize human genetic diversity at population scale, projects such as the 1000 Genomes Project[37], and the large scale sequencing of the Icelandic population[86].

2.1.3 Genomics Studies

2.1.4 Cancer Genomics

Cancer is a genetic disease that has an extremely high burden on the human population. In 2012, the global incidence of new cases worldwide has been estimated as 14.1 million, and deaths at 8.2 million[213]. The economic cost of cancer to the European Union has been estimated at 126 billion euro in 2009[139], and in the US \$124.5 billion USD in 2010[235]. Because of the genetic nature of the disease studying genomes of cancer patients helps uncover the mechanisms behind the development and evolution of cancer[207].

Cancerous tumours arise from a single cell which over time accumulates a series of somatic mutations that cause it to exhibit properties such as: increased mutation rate, increased proliferation, anchorage independent growth, and resisting cell death[88] . Only certain mutations, however, contribute to the development of cancer, while others are benign. Cancer genomics studies aim to identify and characterize those mutations that are cancer drivers and play a role in the formation or progression of tumours[207].

Studying cancer genomes is more complex and expensive than studying the genomes of healthy individuals because each patient requires that two DNA samples are collected - that of the normal tissue, and that of the tumour. This is necessary to identify those mutations that are somatic - i.e. only occur in the tumour cell population[182].

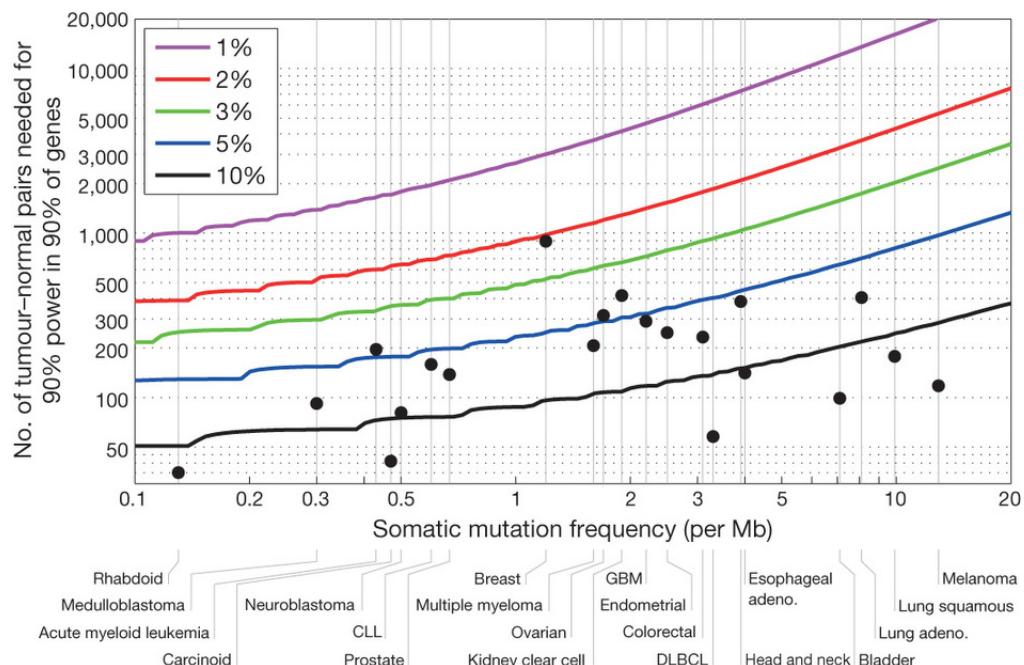


Figure 2.2: Sequencing sample size required by mutation rate[116].

Although there is a large number of identified mutations that are implicated in

cancer (2,002,811 SNV, 10,534 gene fusions, 61,299 genome rearrangements, 695,504 CNV segments in COSMIC v70; August 2014)[71], each mutation has a low chance of being present in any given tumour. Figure 2.2 demonstrates the sample size required to have 90% statistical power to identify 90% of the variants that occur with a set frequency in tumours with varying background mutation rates. Thus, identifying 90% of the mutations occurring with a frequency of at most 1% in Lung Adenocarcinoma requires a sample size of at least 10,000 patients. The necessity to sequence large cohorts of patients in order to be able to comprehensively detect cancer related genomic variants has led to the creation of several large scale cancer sequencing studies.

2.2 Computational Methods for Next Generation Sequencing

Because the size of a typical genome is millions to billions of basepairs long, and current DNA sequencing technology frequently generates errors during the sequencing process, requiring multiple samples of each genomic location to be generated, the amount of data required to be examined in order to characterize even a single sample is well beyond the capabilities of any human. Thus, a multitude of computational approaches are required in order to make the task tractable for individual samples as well as cohorts, and entire populations.

The task of comprehensive characterization of genomic data for an individual is typically decomposed into a series of computational steps, each with its own data representation, and typically developed by a separate research group, which are then assembled into computational pipelines and executed by workflow engines on diverse computing environments. Our goal in this section is to enumerate and describe the individual steps and to provide a survey of the key computational tools and data formats that presently form the set of best practices in this rapidly evolving branch of science. Since Rheos is designed to improve upon these best practices we identify in each section the key mathematical and algorithmic ideas that underpin each approach in order to adapt and translate them into the Rheos framework.

The data that is used in virtually all modern genomics studies is generated on a next generation DNA sequencing machine. Several types of sequencers have been developed but the most frequently used ones are made by Illumina. The raw data produced by such a sequencer is a set of image files, where the color of each pixel represents the corresponding nucleotide base in a DNA strand that is being sequenced in each micro-well of a flowcell, representing the sample of interest. The succession of images produced by each cycle of sequencing then results in a set of reads, a collection of randomly ordered DNA fragments that are further analyzed by downstream tools. The first challenge in generating these reads is the accurate interpretation of pixel colors and mapping them to the corresponding nucleotide bases, known as base-calling. Because all of the currently available DNA sequencing methodologies are imperfect at reading the underlying DNA sequence a number of errors is introduced into the process at various stages and special QA software is

required in order to detect and assess the location and severity of the errors. A typical output of the QA process is a filtered set of reads where the lowest quality reads have been filtered out and each base within each read is assigned a quality score which represents the best current estimate of the probability that the base has been called incorrectly. The currently most frequently used file format for storing DNA sequence reads along with their read qualities is a text file known as fastq.

Depending on whether the organism under study has previously been sequenced there may already exist a reference sequence for it i.e. a file that for each genomic location describes the most frequently occurring nucleotide for that species at that location. Humans, and many other species of organisms already have reference sequences available. If the reference sequence for the organism under study is available then the next processing step involves searching for the position in the reference sequence that best matches each read that has been generated for the sample under study in the previous step. The coordinate of the best match is then assumed to be the location in the genome where that particular read has originated from. This process is known as genome alignment and it is very resource intensive for species with large genomes such as humans (3 billion bases) because a typical sequencing effort will generate at least 1 billion reads for a single sample, and each read needs to be mapped to the reference genome. This problem is made more difficult by the fact that an organism's genome typically has a large proportion of repeated sequence fragments and thus the generated reads do not uniquely align to a single location on the reference. A list of matching positions is generated instead, where each match needs to be scored and the highest scoring match is assumed to be the true origin of the read. Many alignment algorithms exist but the most accurate and fast ones use a two step process of indexing, implemented via hash tables or prefix/suffix tries, to generate a short list of promising match locations, followed by a more exact local alignment that uses dynamic programming to generate a best match. The alignment process is further complicated by the presence of sequencing errors, various genomic variants, and disease state such as cancer, all of which generate significant (and sometimes drastic) differences between the obtained reads and the reference genome, thus necessitating inexact matching approaches. The best algorithms that are currently available have a typical runtime of 24-48 hours on a modern 8-core machine. The most widely adopted standard for storing the alignment data on disk is the SAM[135] (and its binary and indexed counterpart BAM) format developed in the context of the 1000 Genomes Project. In addition to the sequence data and base qualities that are already available in fastq, the SAM format adds a reference coordinate to each read, an overall mapping quality for the read, and whether each position in the read matches the reference sequence, along with other useful metadata.

When a reference sequence does not exist, or when it is undesirable to use one, genome alignment tools are inapplicable and a different approach, called de-novo assembly, is used. Under this approach each read is broken into smaller subsequences called k-mers (of length k), these k-mers are then used to build a graph structure called a de Bruijn graph. Unique paths through the graph represent possible arrangements of reads that correspond to the underlying sequence and the highest scoring path is chosen as the true sequence. Using the de-novo assembly approach

has some advantages over alignment-based methods because it models the structure of the organism's genome directly as it is observed rather than in relation to a reference. This is because no reference is perfect, but instead each reference has its own set of errors that were introduced in its construction. Furthermore, genomic structural variants, which represent large (hundreds to millions of basepairs long) sequences that may be deleted, duplicated, or inverted within a given genome challenge alignment software because of the alignment errors that they introduce and require sophisticated algorithms to later detect, whereas in the de-novo assembly approach these variants are directly modelled as they occur in the underlying sequence and are thus easier to identify. De-novo assembly has its own set of challenges however related to difficulties dealing with repetitive sequences that are found within the genome, as well as the extremely high resource requirements of de-novo assembly algorithms, especially when it comes to memory. The de Bruijn graph is typically built in memory and can be multiple terabytes in size, thus requiring computers with extremely high memory to process. Since, even when using in-memory graph construction the runtime for a single sample is typically several days, it is impractical to move the graph representation to disk without dramatically increasing the algorithm runtime to the point where its duration becomes unreasonable. In practice whole genome de-novo assembly is currently rarely used for processing human genomic data because of the challenges described above. Instead, modern algorithms supplement read alignment with local assembly of particular genomic regions of interest in order to reap some of the benefits offered by assembly-based methods without incurring all of the costs.

Once the reads have been aligned they are typically sorted by genomic coordinate so that all of the reads that overlap a given coordinate can be examined together at once. This is an expensive sortation step that does not lend itself well to parallelization and takes several hours to complete per sample. Subsequent to the sortation step is another round of data QA which aims to throw out low quality reads that poorly align to the reference. Care must be taken however, because these low quality reads may not only signal underlying data or sequencing issues like sample contamination, or lane-swap, but may also signal the presence of structural variants or integration of retrovirus DNA into the host under study, both of which are of high interest to properly identify. Thus, it is common to split the sample into reads of high quality that are further assessed with one set of algorithms and a set of reads that map with low quality, or fail to map at all, to be assessed with a different set of algorithms.

At this point the data is ready to begin the process of variant calling, that is, identifying the genomic features of the sample that are different from the reference sequence for that organism (i.e. mutations). It is important to distinguish germline variant calling from somatic variant calling at this time. In germline variant calling we are trying to identify the set of variants that have been passed to the individuals under study from their parents and are thus present in every cell of the organism forming the underlying genetic background of that individual where some variants may be neutral to the organism's survival, some may be beneficial, and some may be deleterious. Comprehensively identifying and classifying these is of significant research and clinical interest as they confer susceptibility or resistance to certain

disease vectors as well as potential medical remedies and may act as biomarkers to predict disease prognosis or response to treatment within the groups of patients that harbour them.

Somatic mutations are those that each individual cell accumulates over its lifetime and they are of especial interest in the context of cancer where a certain set of mutations accumulated in a particular sequence and over a period of time disrupt the normal cell lifecycle and result in the formation of a malignant tumour. In this context researchers typically sequence both healthy cells (such as those drawn from the patient's blood) and cancerous cells. Mutations are identified in both and the difference between these sets of mutations is then stipulated to be the set of somatic mutations present within that tumour. Just like in the germline case, not all of the somatic mutations contribute to the formation of the cancer and the appropriate identification and classification of those mutations that do (so-called cancer drivers) is an important question of significant clinical and research importance which we consider further below. From a technical standpoint calling somatic variants is significantly more complex than calling germline variants because healthy cells generally conform to the underlying genetic characteristics of the organism, such as the number of chromosomes and ploidy (23 chromosomes, diploid, for humans), whereas in the cancer cells these characteristics can be severely disrupted with entire chromosomes missing or present in amplified copy number, requiring different and more complex statistical models to accurately identify. An additional complexity that is unique to somatic variant calling is the concept of sub-clonal mutations. These are mutations that have been acquired only by some of the cells within a tumour. Since sequencing samples data from a large number of cells within a tumour the reads from which are all pooled together, only a comparatively low number of reads will contain information about sub-clonal mutations, thus making them more difficult to detect, even though such mutations may have a significant impact on the tumour phenotype and thus would be very important to properly identify.

We typically think of three classes of genomic variants that are identified by different methods and oftentimes by separate tools. The simplest to accurately detect, and most frequently occurring are Single Nucleotide Polymorphisms (SNPs), in the germline case, and Single Nucleotide Variants (SNVs), in the somatic case. These are single basepair substitutions where the germline genome differs from the reference sequence by a single letter (for SNPs), or the somatic genome differs from the germline genome by a single letter (for SNVs). SNPs are quite common in humans and occur at the rate of approximately 1 per 1,000 bases on average, or, equivalently, 3 million per individual. Somatic SNVs have a widely varying incidence rate depending on the type of cancer involved with typical rates between (INSERT RATES HERE). For humans, which are diploid (i.e. have two copies of each of the chromosomes, except for the sex chromosomes X and Y), we classify SNPs and SNVs as being either heterozygous (with one reference allele and one variant allele) or homozygous (with both alleles being variant). Methods to detect and accurately genotype SNPs and SNVs typically rely on counting the reads that overlap a given genomic position and evaluating a statistical model that contrasts the probability of the site being reference versus the probability of the site being variant in the face of potential sequencing errors which are expressed as base quality scores and

mapping quality scores (as previously described). The models employed for somatic SNV detection and genotyping are significantly more complex than the models for germline variant detection because of the possibility of sub-clonal mutations (as previously described) as well as regions of amplified copy number (i.e. regions where the organism is no longer diploid but can have any number of additional copies of a chromosomal region, or an entire chromosome). More advanced methods output not only lists of variant sites for a sample but calculate a distribution of genotype likelihoods, i.e. all the possible genotypes at a given variant site along with their relative probabilities so that these can be integrated into the models of downstream statistical analyses in a comprehensive manner.

Indels represent sequence insertions and deletions that are anywhere from 1 base-pair (bp) to about 50 basepairs long. There is no strict upper bound on the length of an indel and individual tools typically decide on their own cutoffs for length although pretty much all tools place their cutoff at a length that is smaller than the typical read length (150 - 500 bp presently). Indel callers typically look for several mismatched bases in a row between the reference and the sample under study and classify the entire length of the mismatched sequence as an insertion or deletion correspondingly. Other indel callers borrow some of the methodology from structural variant callers which are similar to indels, only typically bigger in size, and are potentially more complex.

Structural variants (SVs) are more large scale genomic rearrangements that occur in both germline and somatic genomes and can have a very drastic effect on the organism's phenotype because they can affect a large number of genes at once, resulting in the loss of function of particular important genes, or the creation of gene fusions where, because of a rearrangement, one gene comes under the programmatic control of another gene thereby disrupting important cellular processes. The most common types of structural variants include insertions, deletions, segmental duplications, inversions, and translocations. Simpler structural variants sometimes combine to produce more complex events that are especially difficult to detect properly. The methods for calling and genotyping of structural variants typically rely on looking at the reads that are deemed low quality for the SNP calling process. These are reads that fail to map to the reference genome, split-reads, which are reads where one part of the read maps to one location on the reference and another part of the read maps to another location, and divergently mapped reads (sequencing is frequently done on read pairs where two ends of a DNA fragment of standard size are sequenced in the opposite directions generating a pair of reads with a standard distance, called insert size, inbetween them), with a shorter or longer than expected insert size. SV callers break down these reads into smaller fragments (k-mers) and attempt to map these k-mers to the reference sequence. The goal is to determine the location of breakpoints, which are positions on the sample genome where a DNA strand break is thought to have occurred as part of the genomic rearrangement that has taken place. Once a list of breakpoints is obtained the algorithm attempts to reconstruct the most likely event sequence that these breakpoints could have arisen from, pairing up adjacent breakpoints that are the result of a sequence deletion, for example. Thus, each pair of breakpoints typically gives rise to a single SV call in the final output of the caller. SV calling is a complex and error-prone process

that generates double-digit false-positive and false-negative rate, especially in the somatic case, where patient genomes can undergo drastic rearrangements as a result of cancer-related processes such as chromothripsis and are thus extremely difficult to resolve with accuracy.

Once variants (SNPs/SNVs, Indels, and SVs) have been comprehensively called, a filtering step is necessary because callers are typically initially tuned for highest sensitivity in order to detect the most variants, thus admitting an increased number of false positive calls. Additionally, because calling of SNPs and SVs typically occurs separately by different tools there can be significant call-set overlap where the SNP caller sees a region as a group of SNPs, whereas the SV caller will see it as a single breakpoint. These overlaps need to be resolved in order to avoid redundant calls. A number of filtering approaches exist, some of which rely on heuristics such as strand bias, or read support to filter out low quality variants. Other filtering approaches rely on curated variant databases or machine learning methods in order to reduce the number of false positive calls. One popular filtering approach involves ensemble calling where several different variant calling methods are used on the same dataset and a variant is excluded unless it is called by multiple tools. These methods are typically able to reduce the false positive rate of the call-set by 5-10% while only nominally affecting the false negative rate.

When a filtered high quality call-set has been prepared it is of interest to determine which of the variants are likely to have an effect on the organism's phenotype and which variants are likely to have no consequence. This is accomplished via variant annotation. The annotation process consults a database of known genes and other genomic elements (promoters, enhancers, etc.) to determine the likely consequence of each variant based on the type of mutation that it represents i.e. a synonymous mutation (that doesn't change the underlying amino acid) is likely to have no phenotypic effect, whereas a stop gain mutation inside the coding region of a known gene may indicate a potential loss of function of that gene and may thus have a considerable effect on the observed phenotype. When annotating somatic mutations it is important to consider known cancer genes and delineate whether mutations are "passengers" or "drivers" depending on whether they are thought to be driving the carcinogenesis process by constitutively activating a cancer gene or deactivating a tumour suppressor, or they are simply acquired as part of the genomic instability that is induced by carcinogenesis. An outcome of the variant annotation process then, is a list of somatic or germline variants accompanied by a designation of the known genomic features that they fall in, along with an assigned functional impact. This is typically the last step of an NGS analysis pipeline after which the variant call-set is considered completed and can be used for any number of downstream analyses depending on the particular research question or clinical application being considered. For instance, the variants may be used as input into a Genome Wide Association Study (GWAS), a Quantitative Trait Locus (QTL) analysis, a rare variant association study, or as input into the computation of a clinical biomarker.

2.2.1 File Formats

DNA sequencing studies generate large amounts of data - a single whole-genome sample sequenced at 30x coverage on a modern Illumina sequencer generates roughly 10^9 reads which are strings of length 150 characters and take 100 GB of space on disk when compressed with gzip. Thus, even a moderately-sized study of several thousand individuals needs to grapple with the efficient management of hundreds of terabytes of data. Because of the size of these datasets data storage, access, and exchange formats play a major role in determining the speed, cost, and efficiency with which large-scale analyses can be undertaken. The field of genomics has developed in bursts associated with the major international projects that have been undertaken in the past 30 years, including the Human Genome Project[111], the HapMap Project[40], and the 1000 Genomes Project[37]. It is the latter project that has given rise to most of the currently adopted file format standards in use today, including FASTA/FASTQ for raw sequence data, SAM/BAM for reference-mapped sequence data, and VCF for representing genomic variants. Because these file formats have become the primary information exchange medium in the field of genomics they have a large influence on software tools and data access patterns that are in common use today and thus warrant a closer look.

FASTA/FASTQ

The FASTA/FASTQ file format is a text file format developed at The Sanger Institute for representing genomic sequencing reads[34]. Each read in the file consists of four lines:

- The first is a unique identifier (that encodes some information about the sequencing process). For example - HWUSI-EAS100R:6:73:941:1973#0/1
- The second is the read sequence itself: $S = \{s_i : s \in \{A, C, G, T, N\}\}$. For example - ACGTCCCCGTCCCTNTCCA
- The third is a + sign acting as a separator.
- The fourth is a set of per-base quality scores, represented on the Phred scale, that represent an estimate of the probability that the base has been called correctly. If the error probability is defined as ϵ then $Q_{phred} = -10 \times \log_{10}(\epsilon)$ and conversely $\epsilon = 10^{-\frac{Q_{phred}}{10}}$. In the actual file Phred scores are represented with ASCII characters from !"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_abcdefghijklmnopqrstuvwxyz{|}~ordered by increasing quality, where ! is the lowest possible quality and ~is the highest.

```

@809R9ABXX101220:5:48:9875:59863/1
TTTCTTTCTTTCTTCCTTTTTTTTTTTTTGAGAAGAGATTCTCACCTCTCCAACGCTCC
+
2?>DCBC=CD>@?BECAD:DBAB?BA>@AA>C?CCBCCB#####
@809R9ABXX101220:5:61:10970:200284/1
TTTCTTTCTTTCTTCCTTTTTTTTTTTGGGATGGGGTTTCCCCCTGTTCCC GGCGG
+
BABEECCCEEDCDCFEDFFEDDDDDDDDDDDDDDDDDDDDDCC#####
@809R9ABXX101220:5:66:9966:82226/2
TCTTTCTTTCTTTCTTCCTTTTTTTTTTTGAATGGGGTTCTCCTCCTTTCCCCGGGT
+
>AB@AADD BBCEDCCCC EEECCCCCCCCCCCCCCCCCA>#####
@809R9ABXX101220:5:1:3553:170934/1
CTTTCTTTCTTCCTTTTTTTTTTTGTAGTGGTGGTCCGTCCTGGTCCCTGCCTGCCTC
+
2B;5>><;=;A@:6:>DAABB=6?AB:6CA6>C7>C#####
@809R9ABXX101220:5:24:11915:38264/1
CTTTCTTTCTTCCTTTTTTTTTGGGATGGGGTTCGCTCCTTCCAGGCTGGGTGC
+
BBCCEECCCDFECCFFEDDDDDDDDDDDDDDDCC#####
@809R9ABXX101220:5:65:20977:60085/2
TTTCTTCCTTTTTTTTTTTGAATTGACGTTCCGACCTGTCGCACGGCTTGCTCGACCGTCGCC
+
=; ;<=/;94??AABAA8BCABB7BA#####
@809R9ABXX101220:5:27:2026:120477/1
TTTTTTTTTTTTGGAGAGGGGTTCGCCCTGTTGCCAGGCTGGACTGCCCGTCCCGGCCAAAAA
+
>>>?@cccccc@cc@cc@AA#####

```

Figure 2.3: Excerpt from a FASTQ file. /1 and /2 at the end of read ID indicate whether read is first or second in pair.

Since FASTQ is a large text file with highly repetitive content it is typically stored in a compressed manner.

SAM/BAM

The Sequence Alignment Map (SAM) text file format and its accompanying binary compressed version BAM was created for sequencing data storage and analysis in the context of the 1000 genomes project[135]. These files provide additional fields on top of the ones available in FASTA/FASTQ in order to relay information related to sequence alignment to a reference genome, and are the most widely used format for storing DNA sequencing data today. A SAM file consists of a header and a body. These are described below. All tables, descriptions, and definitions in this section are reproduced or adapted from the SAM file specification at <https://sam-tools.github.io/hts-specs/SAMv1.pdf>.

SAM Header Each line of the header begins with the character ‘@’ followed by a header record (see Table 2.1). Each line is TAB-delimited and, apart from @CO lines, each data field follows the format ‘TAG:VALUE’ where TAG is a two-character string that defines the format and content of VALUE.

Table 2.1: SAM file header record and column definition. Tags listed with ‘*’ are required. (adapted from <https://samtools.github.io/hts-specs/SAMv1.pdf>)

Tag	Description
©HD	The header line. The first line if present.
VN*	Format version. Accepted format: $/^[\text{0-9}]+\.\text{[0-9]}+$/$.
SO	Sorting order of alignments. Valid values: <code>unknown</code> (default), <code>unsorted</code> , <code>queryname</code> and <code>coordinate</code> .
GO	Grouping of alignments, indicating that similar alignment records are grouped together but the file is not necessarily sorted overall. Valid values: <code>none</code> (default), <code>query</code> (alignments are grouped by QNAME), and <code>reference</code> (alignments are grouped by RNAME/POS).
SS	Sub-sorting order of alignments. Valid values are of the form <i>sort-order</i> : <i>sub-sort</i> , where <i>sort-order</i> is the same value stored in the SO tag and <i>sub-sort</i> is an implementation-dependent colon-separated string further describing the sort order. Regular expression: <code>(coordinate queryname unsorted)(:[A-Za-z0-9_-]+)+</code>
©SQ	Reference sequence dictionary. The order of ©SQ lines defines the alignment sorting order.
SN*	Reference sequence name. The SN tags and all individual AN names in all ©SQ lines must be distinct. The value of this field is used in the alignment records in RNAME and RNEXT fields. Regular expression: <code>[!-)+-<>~-]![~-]*</code>
LN*	Reference sequence length. Range: $[1, 2^{31} - 1]$
AH	Indicates that this sequence is an alternate locus.
AN	Alternative reference sequence names.
AS	Genome assembly identifier.
DS	Description. UTF-8 encoding may be used.
M5	MD5 checksum of the sequence.
SP	Species.
UR	URI of the sequence.
©RG	Read group. Unordered multiple ©RG lines are allowed.
ID*	Read group identifier. Each ©RG line must have a unique ID.
BC	Barcode sequence identifying the sample or library.
CN	Name of sequencing center producing the read.
DS	Description. UTF-8 encoding may be used.
DT	Date the run was produced (ISO8601 date or date/time).
FO	Flow order. The array of nucleotide bases that correspond to the nucleotides used for each flow of each read.
KS	The array of nucleotide bases that correspond to the key sequence of each read.
LB	Library.
PG	Programs used for processing the read group.
PI	Predicted median insert size.
PL	Platform/technology used to produce the reads. Valid values: CAPILLARY, LS454, ILLUMINA, SOLID, HELICOS, IONTORRENT, ONT, and PACBIO.
PM	Platform model. Free-form text providing further details of the platform/technology used.
PU	Platform unit (e.g. flowcell-barcode.lane for Illumina or slide for SOLiD). Unique identifier.
SM	Sample. Use pool name where a pool is being sequenced.
©PG	Program.
ID*	Program record identifier. Each ©PG line must have a unique ID. The value of ID is used in the alignment PG tag and PP tags of other ©PG lines. PG IDs may be modified when merging SAM files in order to handle collisions.
PN	Program name
CL	Command line. UTF-8 encoding may be used.
PP	Previous ©PG-ID. Must match another ©PG header’s ID tag. ©PG records may be

Table 2.3: SAM file alignment record mandatory column definition. (adapted from <https://samtools.github.io/hts-specs/SAMv1.pdf>)

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	[!-?A-~]{1,254}	Query template NAME
2	FLAG	Int	[0, 2 ¹⁶ - 1]	bitwise FLAG
3	RNAME	String	* [!-()+-<>~-] [!-~]*	Reference sequence NAME
4	POS	Int	[0, 2 ³¹ - 1]	1-based leftmost mapping POSition
5	MAPQ	Int	[0, 2 ⁸ - 1]	MAPping Quality
6	CIGAR	String	* ([0-9]+[MIDNSHPX=])+	CIGAR string
7	RNEXT	String	* = [!-()+-<>~-] [!-~]*	Ref. name of the mate/next read
8	PNEXT	Int	[0, 2 ³¹ - 1]	Position of the mate/next read
9	TLEN	Int	[-2 ³¹ + 1, 2 ³¹ - 1]	observed Template LENgth
10	SEQ	String	* [A-Za-z.=.]+	segment SEQuence
11	QUAL	String	[!-~]+	ASCII of Phred-scaled base QUALity+33

SAM Body The body of a SAM file contains alignment records (see Section 2.2.2 on details of alignment algorithms). Each record has 11 mandatory fields. These fields always appear in the same order and must be present, but their values may be ‘0’ or ‘*’ (depending on the field) if the corresponding information is unavailable. Table 2.3 lists the mandatory fields in the SAM format: Alignment records represent the information contained in a sequencing read subsequent to it being aligned to a reference genome (see Figure 2.4).

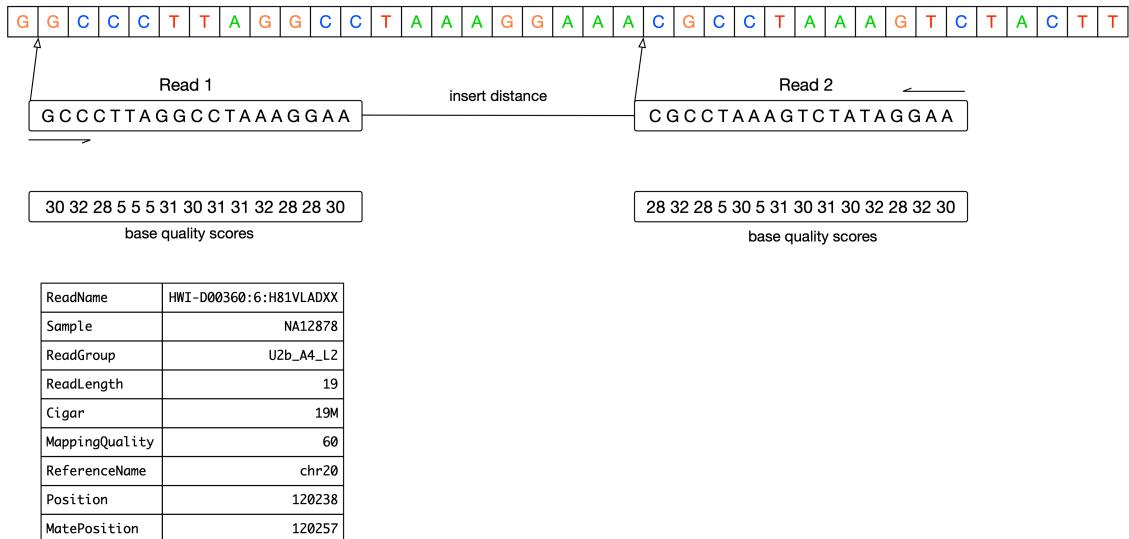


Figure 2.4: A read-pair that is aligned to the reference.

1. **QNAME:** Query template NAME. Reads/segments having identical QNAME are regarded to come from the same template. A QNAME ‘*’ indicates the information is unavailable. In a SAM file, a read may occupy multiple alignment lines, when its alignment is chimeric or when multiple mappings are given.
2. **FLAG:** Combination of bitwise FLAGS. Each bit is explained in the following table:

Bit	Description
1	0x1 template having multiple segments in sequencing
2	0x2 each segment properly aligned according to the aligner
4	0x4 segment unmapped
8	0x8 next segment in the template unmapped
16	0x10 SEQ being reverse complemented
32	0x20 SEQ of the next segment in the template being reverse complemented
64	0x40 the first segment in the template
128	0x80 the last segment in the template
256	0x100 secondary alignment
512	0x200 not passing filters, such as platform/vendor quality controls
1024	0x400 PCR or optical duplicate
2048	0x800 supplementary alignment

- For each read/contig in a SAM file, it is required that one and only one line associated with the read satisfies ‘FLAG & 0x900 == 0’. This line is called the *primary line* of the read.
- Bit 0x100 marks the alignment not to be used in certain analyses when the tools in use are aware of this bit. It is typically used to flag alternative mappings when multiple mappings are presented in a SAM.
- Bit 0x800 indicates that the corresponding alignment line is part of a chimeric alignment. A line flagged with 0x800 is called as a *supplementary line*.
- Bit 0x4 is the only reliable place to tell whether the read is unmapped. If 0x4 is set, no assumptions can be made about RNAME, POS, CIGAR, MAPQ, and bits 0x2, 0x100, and 0x800.
- Bit 0x10 indicates whether SEQ has been reverse complemented and QUAL reversed. When bit 0x4 is unset, this corresponds to the strand to which the segment has been mapped. When 0x4 is set, this indicates whether the unmapped read is stored in its original orientation as it came off the sequencing machine.
- Bits 0x40 and 0x80 reflect the read ordering within each template inherent in the sequencing technology used.¹ If 0x40 and 0x80 are both set, the read is part of a linear template, but it is neither the first nor the last read. If both 0x40 and 0x80 are unset, the index of the read in the template is unknown. This may happen for a non-linear template or when this information is lost during data processing.
- If 0x1 is unset, no assumptions can be made about 0x2, 0x8, 0x20, 0x40 and 0x80.
- Bits that are not listed in the table are reserved for future use. They should not be set when writing and should be ignored on reading by current software.

¹For example, in Illumina paired-end sequencing, **first** (0x40) corresponds to the R1 ‘forward’ read and **last** (0x80) to the R2 ‘reverse’ read. (Despite the terminology, this is unrelated to the segments’ orientations when they are mapped: either, neither, or both may have their reverse flag bits (0x10) set after mapping.)

3. **RNAME**: Reference sequence NAME of the alignment. If @SQ header lines are present, **RNAME** (if not ‘*’) must be present in one of the **SQ-SN** tag. An unmapped segment without coordinate has a ‘*’ at this field. However, an unmapped segment may also have an ordinary coordinate such that it can be placed at a desired position after sorting. If **RNAME** is ‘*’, no assumptions can be made about **POS** and **CIGAR**.
4. **POS**: 1-based leftmost mapping POSition of the first **CIGAR** operation that “consumes” a reference base (see table below). The first base in a reference sequence has coordinate 1. **POS** is set as 0 for an unmapped read without coordinate. If **POS** is 0, no assumptions can be made about **RNAME** and **CIGAR**.
5. **MAPQ**: MAPping Quality. It equals $-10 \log_{10} \Pr\{\text{mapping position is wrong}\}$, rounded to the nearest integer. A value 255 indicates that the mapping quality is not available.
6. **CIGAR**: CIGAR string. The CIGAR operations are given in the following table (set ‘*’ if unavailable):

Op	BAM	Description	Consumes query	Consumes reference
M	0	alignment match (can be a sequence match or mismatch)	yes	yes
I	1	insertion to the reference	yes	no
D	2	deletion from the reference	no	yes
N	3	skipped region from the reference	no	yes
S	4	soft clipping (clipped sequences present in SEQ)	yes	no
H	5	hard clipping (clipped sequences NOT present in SEQ)	no	no
P	6	padding (silent deletion from padded reference)	no	no
=	7	sequence match	yes	yes
X	8	sequence mismatch	yes	yes

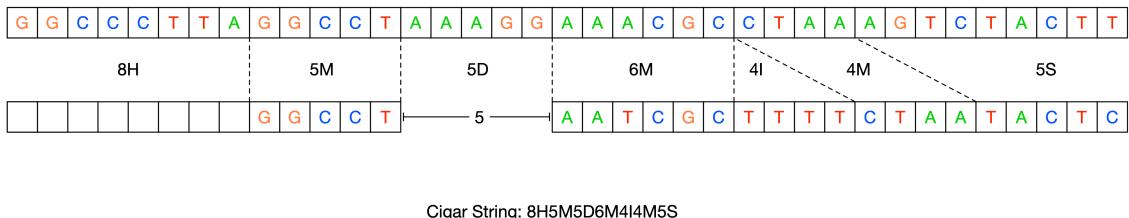


Figure 2.5: Example of an alignment CIGAR string.

- “Consumes query” and “consumes reference” indicate whether the CIGAR operation causes the alignment to step along the query sequence and the reference sequence respectively.
- H can only be present as the first and/or last operation.
- S may only have H operations between them and the ends of the CIGAR string.

- For mRNA-to-genome alignment, an N operation represents an intron. For other types of alignments, the interpretation of N is not defined.
 - Sum of lengths of the M/I/S/=/X operations shall equal the length of SEQ.
7. RNEXT: Reference sequence name of the primary alignment of the NEXT read in the template. For the last read, the next read is the first read in the template. If @SQ header lines are present, RNEXT (if not '*' or '=') must be present in one of the SQ-SN tag. This field is set as '*' when the information is unavailable, and set as '=' if RNEXT is identical RNAME. If not '=' and the next read in the template has one primary mapping (see also bit 0x100 in FLAG), this field is identical to RNAME at the primary line of the next read. If RNEXT is '*', no assumptions can be made on PNEXT and bit 0x20.
 8. PNEXT: 1-based Position of the primary alignment of the NEXT read in the template. Set as 0 when the information is unavailable. This field equals POS at the primary line of the next read. If PNEXT is 0, no assumptions can be made on RNEXT and bit 0x20.
 9. TLEN: signed observed Template LENGTH. If all segments are mapped to the same reference, the unsigned observed template length equals the number of bases from the leftmost mapped base to the rightmost mapped base. The leftmost segment has a plus sign and the rightmost has a minus sign. The sign of segments in the middle is undefined. It is set as 0 for single-segment template or when the information is unavailable.
 10. SEQ: segment SEQuence. This field can be a '*' when the sequence is not stored. If not a '*', the length of the sequence must equal the sum of lengths of M/I/S/=/X operations in CIGAR. An '=' denotes the base is identical to the reference base. No assumptions can be made on the letter cases.
 11. QUAL: ASCII of base QUALity plus 33 (same as the quality string in the Sanger FASTQ format). A base quality is the phred-scaled base error probability which equals $-10 \log_{10} \Pr\{\text{base is wrong}\}$. This field can be a '*' when quality is not stored. If not a '*', SEQ must not be a '*' and the length of the quality string ought to equal the length of SEQ.

BAM Files BAM files are SAM files that have been compressed with block gzip compression[128]. If the reads inside the BAM file are sorted in increasing order of reference coordinate, the BAM file supports random access via a supplementary BAM index (*.bam.bai) file. Each block inside a BAM file is a separate gzip archive up to 64Kb in size with extra metadata to encode the read positions contained inside the block. The BAM index file contains offsets into the BAM file that correspond to particular genomic coordinate ranges. The BAM file this provides a binary compressed structure that supports indexed search on genomic coordinates. No other indexing, including by ID, is currently supported. The majority of DNA sequencing data in the world is currently stored as BAM files.

VCF Files

The Variant Call Format (VCF)[49] is a type of text file that was created in the context of the 1000 Genomes project to allow the representation of various types of genetic variation that are discovered via NGS experiments. VCF files describe variants in a reference-relative manner - i.e. all genetic variation is shown with respect to a given reference genome build. VCF files have a tabular form and allow one to describe genetic variants as they occur in (or are absent from) a entire cohort of samples using a single file. This file format, despite several significant limitations, has become the widely adopted standard for representing genetic variation. All descriptions in this section have been reproduced or adapted from the 2011 Danecek et al. paper and the VCF specification ()

A VCF file consists of the following sections - Meta-information section describes the format and content of the data contained in the file, header section specifies the columns present in the file, data section contains the actual variant calls that are being described.

Meta-information The meta-information section contains a number of lines that describe the data section. The first line of this section is a *fileformat* line that specifies what version of the VCF spec the file adheres to. There can be any number of lines describing INFO fields. These fields are populated into the INFO column of the data section. A single INFO field may describe a single value or a tuple of values. These values can be Integer, Float, Flag, Character, or String. All of the INFO fields thus described are to be placed into the INFO column as a string of semi-colon-separated key-value pairs. The FILTER field describes any filters that have been applied to the variants. The FORMAT field describes the format of the genotype columns that are specified in the data section. The ALT field describes symbolic alternate alleles, that result from variants that have been called but not accurately genotyped. The *contig* field lists the contigs that the variants specified in the file have been called relative to (typically these are chromosome names from the reference sequence). The SAMPLE field specifies the samples that the variants map to. The PEDIGREE field specifies pedigree relationships between samples in the file.

Header The header section is a single tab-delimited line that lists what columns are present in the body. The mandatory columns are:

- #CHROM
- POS
- ID
- REF
- ALT

- QUAL
- FILTER
- INFO

If the VCF file contains genotypes then the INFO column is followed by a FORMAT column, followed by one column for each present sample where each column name is the respective sample ID and all sample IDs are unique within the file.

Data The data section of a VCF file contains the actual list of variants and their genotypes in all samples in tabular form where the columns align with the header section. The values are tab-separated. Any missing values are indicated with a ." (dot). The line contents are as follows:

CHROM - An identifier of the chromosome where the variant resides. The ID of the chromosome should match one of the *contig* entries in the meta-information section. All of the variants that belong to a single CHROM should exist in a single contiguous block of rows in a VCF file.

POS - 1-based position of the variant with respect to the reference chromosome specified in CHROM. Variant positions should be sorted numerically in increasing order.

ID - A semi-colon separated list of unique identifiers, where they exist. If no identifier exists a missing value should be indicated.

REF - the reference bases corresponding to the variant location where each base $b \in \{A, C, G, T, N\}$.

ALT - a comma separated list of alternative alleles. The alleles do not have to called in any of the samples. Value can be either a String of $b \in \{A, C, G, T, N, *\}$ or a missing value (when there is no variant).

QUAL - the PHRED scale variant quality. When ALT is present QUAL is $-10 \log_{10}(\text{no variant})$ and when ALT is missing QUAL is $-10 \log_{10}(\text{variant})$. If QUAL is unknown then missing value must be specified.

FILTER - encodes the filter status. If the variant passes all QC filters the value should be PASS. Otherwise there should be a semi-colon separated list of codes for filters that have not passed. If FILTER information is not available there should be a missing value.

INFO - A list of key-value pairs for the additional fields encoded for each variant as specified in the INFO lines of the metadata section. Keys without corresponding values may be used to indicate group membership. There is a number of frequently used reserved keys (see Table 2.4).

Key	Number	Type	Description
AA	1	String	Ancestral allele
AC	A	Integer	Allele count in genotypes, for each ALT allele, in the same order as listed
AD	R	Integer	Total read depth for each allele
ADF	R	Integer	Read depth for each allele on the forward strand
ADR	R	Integer	Read depth for each allele on the reverse strand
AF	A	Float	Allele frequency for each ALT allele in the same order as listed (estimated from primary data, not called genotypes)
AN	1	Integer	Total number of alleles in called genotypes
BQ	1	Float	RMS base quality
CIGAR	A	String	Cigar string describing how to align an alternate allele to the reference allele
DB	0	Flag	dbSNP membership
DP	1	Integer	Combined depth across samples
END	1	Integer	End position (for use with symbolic alleles)
H2	0	Flag	HapMap2 membership
H3	0	Flag	HapMap3 membership
MQ	1	Float	RMS mapping quality
MQ0	1	Integer	Number of MAPQ == 0 reads
NS	1	Integer	Number of samples with data
SB	4	Integer	Strand bias
SOMATIC	0	Flag	Somatic mutation (for cancer genomics)
VALIDATED	0	Flag	Validated by follow-up experiment
1000G	0	Flag	1000 Genomes membership

Table 2.4: Reserved INFO keys (from <https://samtools.github.io/hts-specs/VCFv4.3.pdf>).

If genotype information is present, the exact same definition is used for all samples. The FORMAT field specifies a column separated list of the data types and their order that are present in the genotype columns. This is followed by one data block for each sample that contains the genotype data as described in the FORMAT column. All keys are optional but missing values should be indicated. There is a number of frequently used and reserved keys (see Table 2.5).

Field	Number	Type	Description
AD	R	Integer	Read depth for each allele
ADF	R	Integer	Read depth for each allele on the forward strand
ADR	R	Integer	Read depth for each allele on the reverse strand
DP	1	Integer	Read depth
EC	A	Integer	Expected alternate allele counts
FT	1	String	Filter indicating if this genotype was “called”
GL	G	Float	Genotype likelihoods
GP	G	Float	Genotype posterior probabilities
GQ	1	Integer	Conditional genotype quality
GT	1	String	Genotype
HQ	2	Integer	Haplotype quality
MQ	1	Integer	RMS mapping quality
PL	G	Integer	Phred-scaled genotype likelihoods rounded to the closest integer
PQ	1	Integer	Phasing quality
PS	1	Integer	Phase set

Table 2.5: Reserved genotype keys (from <https://samtools.github.io/hts-specs/VCFv4.3.pdf>).

The following keys are most important and frequently used:

GT - The genotype, encoded as allele values separated by / for unphased genotypes and | for phased genotypes. The values are 0 for reference allele, 1 for first allele listed in ALT, 2 for the second, and so on (for instance 0/1 for heterozygous variant in a diploid sample). When a call cannot be made the missing value is used (for instance ./).

GL - Genotype likelihoods. A comma separated list of \log_{10} likelihoods for all possible genotypes given the set of REF and ALT alleles at the locus.

GP - Genotype posterior probabilities, in the same order as GL field.

GQ - Genotype Quality in PHRED scale. Probability the genotype call is wrong given that the site is variant.

AD - Allele Depth. Per-sample read depth for each allele.

DP - Read Depth. $\sum_i AD_i$.

See Listing 33 for an example of a VCF file.

Remarks

The SAM/BAM and VCF file formats have become the de-facto standards for representing sequencing read data and genetic variation respectively. Because of this status most computational tools that exist in this space either consume or produce one of these data-types and the data analysis process is influenced by the file access patterns inherent in these standards. This introduces a number of challenges and limitations that have held back scalability of the existing tools to larger data sets. Specifically:

- The focus on files for information storage and retrieval implies that sophisticated file management schemes must be deployed for successful management of large cohorts of samples. This includes concerns of data security, and data migration that must be implemented at the file-system level.
- Storage of sequence data at sample level of granularity in SAM/BAM files creates very large files that are typically greater than 150 GB in size per sample and are thus quite cumbersome to work with.
- Lack of usage of databases implies only basic indexing and querying schemes are possible for sequencing and variant data. Large amounts of data thus need to be loaded into memory in order to perform basic queries.
- Indexing only by genomic coordinate implies a coordinate-based data traversal mechanisms that process a genome linearly from beginning to the end.
- The multi-sample VCF format makes it easy to interpret all carriers for a single variant (by reading a single row), but is not well suited for interrogating all variants for a single individual (scanning all rows for a single column) in a text file.
- Storage of sequencing data in multiple per-sample files does not take advantage of extreme sequence similarity between samples at a given genomic locus, thus reducing the opportunities for data compression and driving up project costs for large scale sequencing efforts where data set size exceeds 1PB.

The stream-based approach adopted by the Rheos framework described in this work attempts to address many of the above issues.

2.2.2 Alignment

Genome alignment (also called mapping) is the process that, given a DNA sequencing read, finds the location in a reference genome that the read best matches to. This is an important process in a next generation sequencing pipeline because it provides a way of ordering the otherwise unordered collection of raw reads (by reference coordinate) that can be used to find locations where the sample genome is different from the reference (see Figure 2.6).



Figure 2.6: Collection of reads aligned to a reference genome and evaluated at a locus that contains a heterozygous SNP.

This process is equivalent to substring search where the string being searched is 3×10^9 characters long and there are 10^9 patterns of length 150 to be found. This process is complicated by the fact that reads from a sample have both sequencing errors and genuine genetic variation that make them differ from the reference, and the fact that certain regions of the genome can be highly repetitive, with the same sequence pattern occurring hundreds of times, making unique mapping challenging.

There are several key applications of alignment that involve sequences of varying length, require different properties, and may not all be best accomplished by the same algorithms. These are:

- Alignment of individual single-end and paired-end reads to a human reference sequence. Where most of the read is expected to match successfully. This is the most abundant use case and the one towards which most alignment algorithms and software implementations are geared.
- Alignment of split-reads to a reference sequence. Split-reads are those where a part of the read maps to one position in the reference sequence and another part of the read maps to a different part of the reference indicating that the read spans a genomic rearrangement. Reads that admit a split alignment may

come out unmapped from a normal alignment stage. This type of alignment is important for both germline structural variant calling and somatic structural variant calling where genomic aberrations are more common.

- Alignment of reads to a group of reference genomes for common pathogens (viruses, bacteria), as well as other common sample contaminating species.
- Alignment of assembled contigs to the reference sequence after local assembly. Some variant calling methods will perform local assembly of reads into a group of potential haplotypes and will perform alignment of these haplotypes to the reference sequence to identify where the actual variants are. The haplotypes may be anywhere from hundreds to millions of bases long.
- Alignment of reads to a group of alternative haplotypes. Variant callers may generate a list of candidate alternative haplotypes and reads need to be aligned to all of these in order to determine which haplotype is best supported by the reads data.

Initially, algorithms for mapping sequencing reads (such as Maq[133] and SOAP[136]) have used a hash table approach[178] (building up tables of subsequences) for finding promising approximate locations for a read within the reference, and then using the Smith-Waterman[202] dynamic programming algorithm for selecting the best matching location. These approaches, however, have proven to be quite slow with typical runtimes exceeding 72 hours on a single high coverage whole genome sample. For the past decade the best available genome alignment approaches in terms of balanced accuracy and speed of processing have been based on the Burrows-Wheeler Transform (BWT)[27] and the FM index[69]. Two of the most popular and widely used are Bowtie[114] and BWA[131]. With recent improvements in computer power, hash table based approaches, such as minimap2[126] and SNAP[240] have made a comeback and are again becoming competitive in the alignment space. A commercial tool called Novoalign (<http://www.novocraft.com/products/novoalign/>) has consistently been a top performer in terms of speed and accuracy, but the method is not publically available.

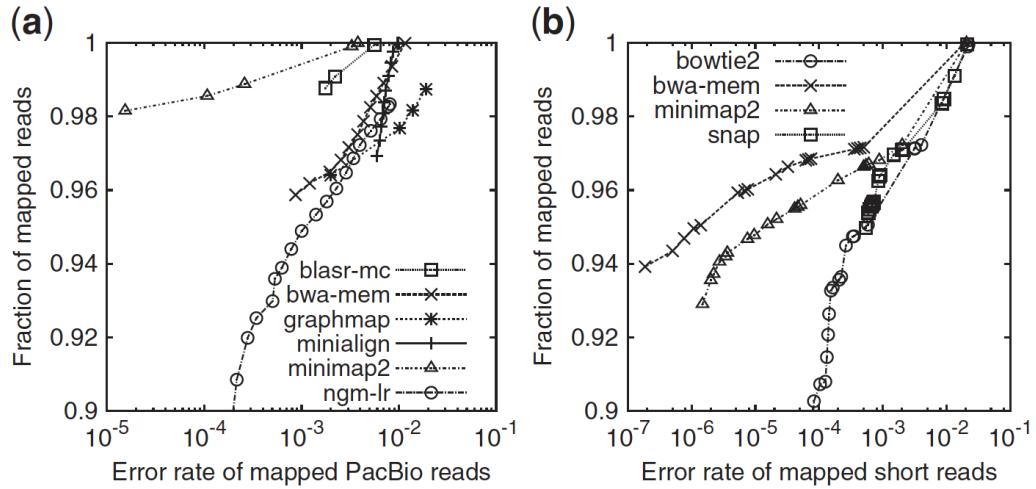


Figure 2.7: Comparison of several aligners on simulated a)long reads >1000 bp, and b) short reads 150bp.[126]

String search is a well studied field, and many genome aligners use the equivalence relationship between suffix trees, suffix arrays, and the BWT (see Figure 2.8) to construct searchable and compressed data structures suitable for genome alignment applications.

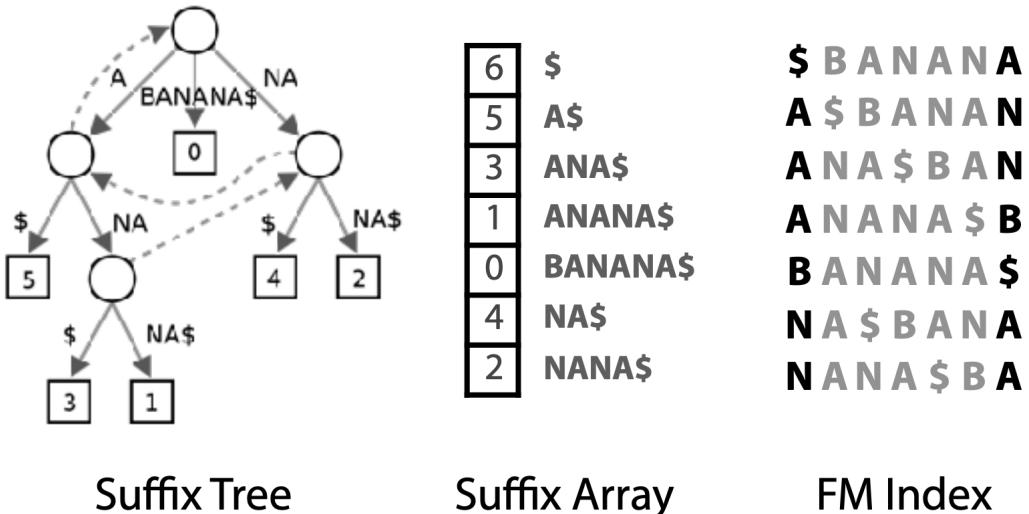


Figure 2.8: Equivalence between a Suffix Tree, Suffix Array, and the BWT matrix of the string BANANA.[1]

Suffix Tree A suffix *trie* T is a tree data structure that given a string $s = c_1, c_2, \dots, c_n$ over an alphabet Σ , such that $c_i \in \Sigma$ encodes all of the suffixes of s from the root to the leaves. s is terminated with a sentinel character $\$$. For any character $c \in \Sigma$ and node $n_i \in T$, n_i has at most one child labeled c . Every leaf is the sentinel character $\$$. See Figure 2.9 for example.

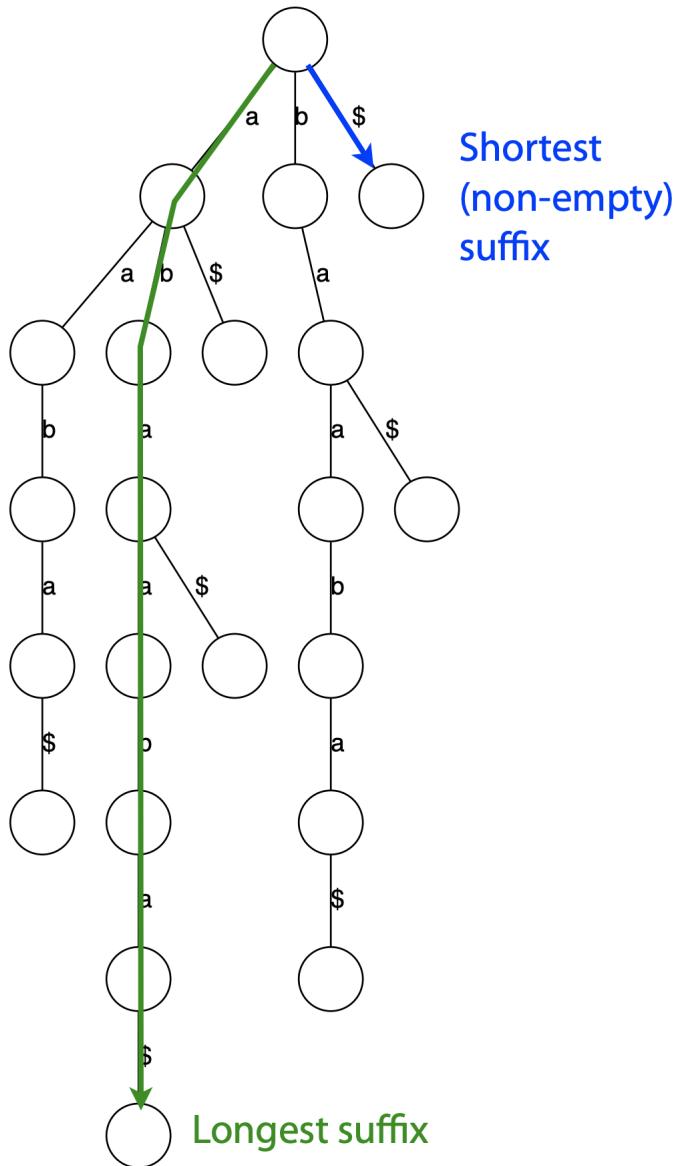


Figure 2.9: Suffix trie of string `abaaba$`.[1]

Because every substring of s is a prefix of some suffix of s , given a search query q one can check whether q is a substring of s in $\mathcal{O}(|q|)$ time by progressively matching each character of q from the root of T until either all of q is matched (hence q is a substring of T) or at some point a character of q does not have a matching node in T , indicating that q is not a substring of T . The size (number of nodes) of the trie can grow with $\mathcal{O}(|s|^2)$

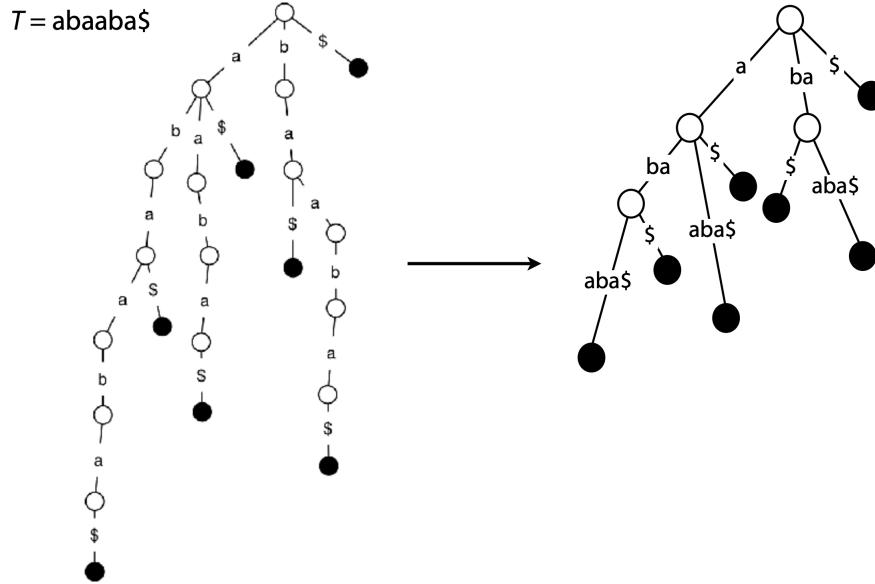


Figure 2.10: Create suffix tree by coalescing non-branching paths into single nodes.[1]

The size of the tree can be reduced by coalescing all non-branching subpaths of T into single nodes (see Figure 2.10).

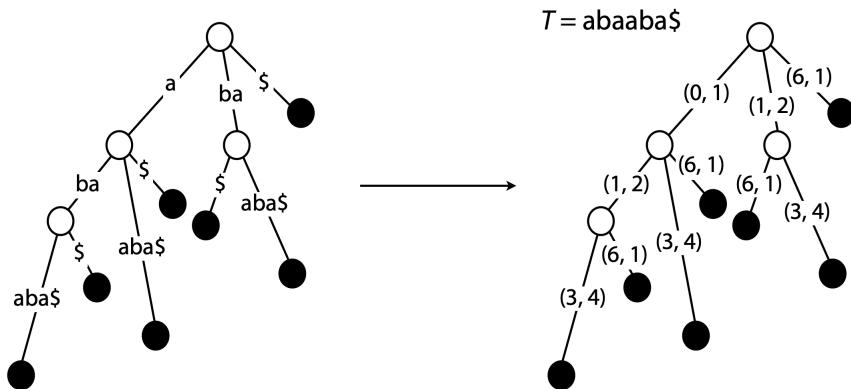


Figure 2.11: Reduce size by replacing substrings by offsets into the original string.[1]

The size can be further reduced to $\mathcal{O}(|s|)$ by replacing string nodes with pairs of offsets into the original string s (see Figure 2.11).

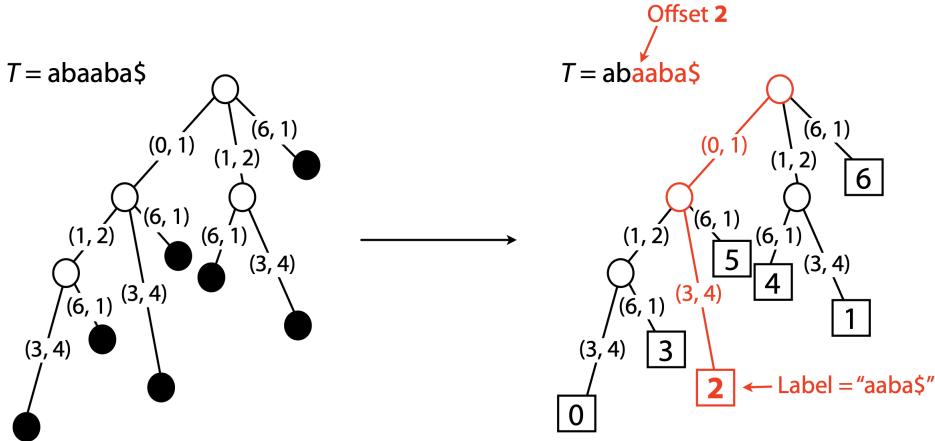


Figure 2.12: Aid search by storing index of substring in leaves.[1]

Searching can be aided by storing offsets of suffixes in the leaf node of the path spelling out that suffix. See Figure 2.12. The online construction algorithm by Ukkonen[215] allows creation of suffix trees in $\mathcal{O}(|s|)$ time and space. Although search against the suffix tree is very fast - $\mathcal{O}(|q|)$, the tree size is still an issue when considering applications to human genome sequencing. A suffix tree for a human would take occupy more than 45 GB of memory[1]. Furthermore, alignment of real reads is made more complex because reads have errors and encode genetic variation, thus inexact searching mechanisms are required.

Suffix Array Suffix arrays were developed by Manber and Myers[141] as a data structure that is equivalent to a suffix tree but occupies less space. Given a string $s = c_1, c_2, \dots, c_n$, and letting $s[i, j]$ be a substring of s between indexes i and j , the suffix array SA of s contains integers providing the starting positions of suffixes of s sorted in lexicographical order, s.t. $\forall i \in [1, n] : s[SA[i - 1], n] < s[SA[i], n]$. See Figure 2.8 for an example suffix array representing the string *banana*. Because the array is sorted, a simple binary search is possible that finds or rules out a match of query q in s in $\mathcal{O}(|q| \log |s|)$, although a more sophisticated search algorithm in $\mathcal{O}(|q| + \log |s|)$ is possible[141]. A suffix array can be constructed by a depth-first traversal of a suffix tree in $\mathcal{O}(|s|)$ time, with the best current algorithm due to Kärkkäinen[105]. Using the array can bring the space requirements for a human-sized genome search index down to about 16GB[1].

The BWT (see Figure 2.13) is constructed from an original string s by appending a special termination character $\$$ that is lexicographically smaller than all other characters and does not occur in s . Then a matrix of all cyclic rotations of $s\$$ is constructed and sorted in ascending lexicographical order (Figure 2.13 a). The last column of this matrix is the BWT, which can be used to reconstruct the original string and perform substring searches on it. This mechanism relies on the fact that all common prefixes of substrings occupy a contiguous block of rows in the BWT matrix (because of lexicographical ordering, Figure 2.13 c)), and the LF (last-first) property of the BWT. Namely, the character in the last column of the BWT directly

precedes the character in the first column of the BWT in the original string. And, the index of the occurrence of a character in the last column of the BWT is the same as the index of the occurrence of the same character in the first column of the BWT (see Figure 2.13 b)) i.e. the first occurrence of *g* in the last column corresponds to the first occurrence of *g* in the first column. The third occurrence of *a* in the last column corresponds to the third occurrence of *a* in the first column, etc.

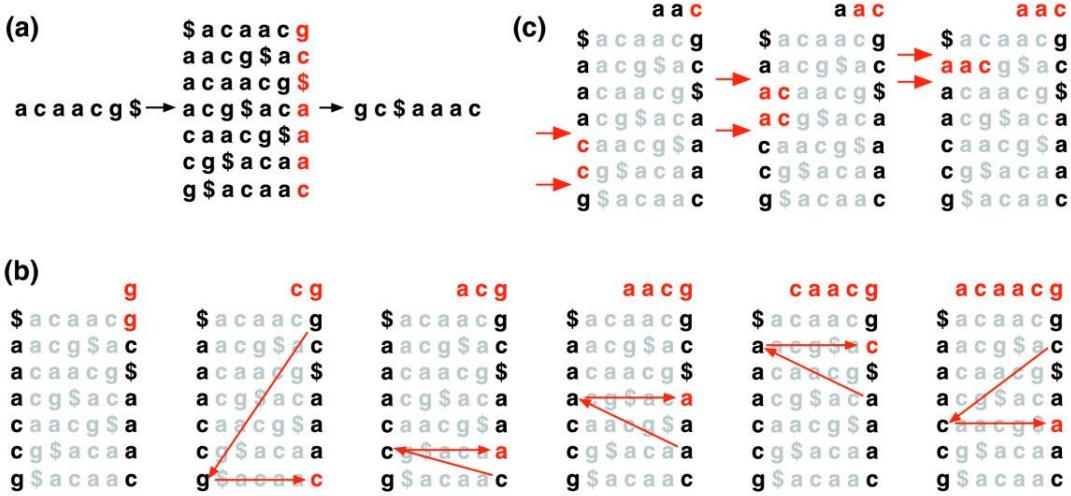


Figure 2.13: a) Creating the BWT of a string. b) All substrings with the same prefix occupy adjacent rows. c) The original string can be recovered from the BWT by using the LF rule.[114]

Alternatively the BWT can be constructed from the suffix array via:

$$BWT[i] = \begin{cases} s[SA[i] - 1], & \text{if } SA[i] > 1 \\ \$, & \text{otherwise} \end{cases} \quad (2.1)$$

, per [199]

The entire original string can be recovered from the BWT in the following manner. The last column is the BWT. The first column is the lexicographically sorted BWT. Find the row that starts with \$. The character in the last column in that row is the last character of original string *s* (*g* in the example above). Find the first occurrence of that character in the first column. Look up the character in the last column of that row (*c* in the example above). This is the second-last character of *s*. Note the index of its occurrence in the last column (it's a second *c* in the example). Find the row with the same character and the same index in the first column (second *c* in the first column). The last character in that row is the second-last character of *s*. Repeat this process until the last character in a row is \$. This recovers the original string.

FM Index

Bowtie Bowtie is a popular aligner developed by Ben Langmead in 2009, with a focus on fast alignment of short reads[114], later followed up and enhanced with Bowtie 2 for longer reads[112], and adapted massively parallel deployment on the AWS cloud with Crossbow[113]. Since the <50bp long short-reads that the original Bowtie was created for are no longer seen we focus our exposition on Bowtie 2.

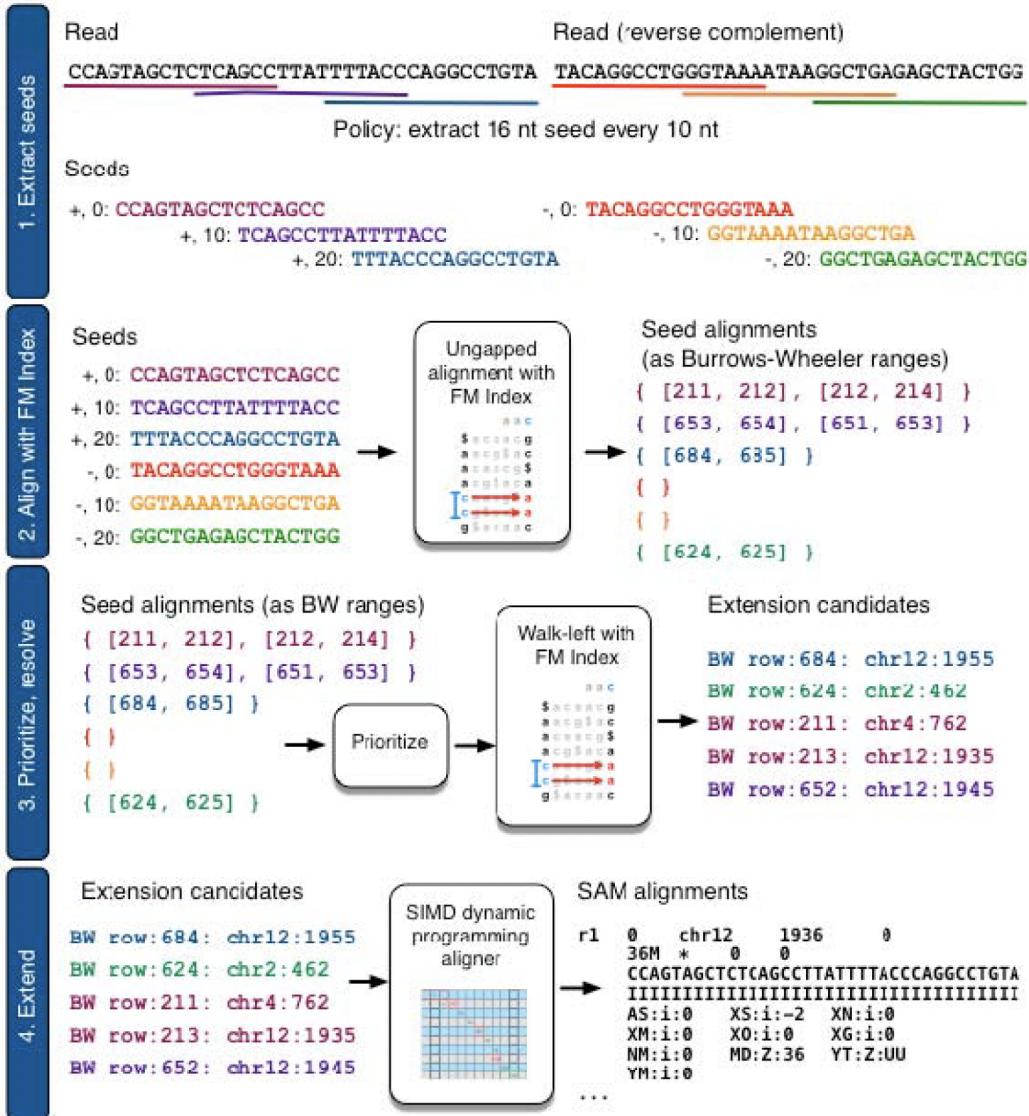


Figure 2.14: Steps used by Bowtie 2 alignment.[112]

Figure 2.14 shows the sequence of steps used by the Bowtie 2 alignment process. One of the big challenges for aligners is the ability to deal with gapped alignment i.e. sequences in the reads that are inserted or deleted with respect to the reference. These may be the result of sequencing errors or genuine genetic variation present in the sample. Since the read sequence cannot be matched to the reference exactly, gapped alignment drastically increases the search space of possible matches that need to be considered, corresponding to different possible gap sizes. In order to mitigate the degrading effect of gapped alignment on performance, Bowtie adopts a seed-

and-extend mechanism where it performs ungapped exact alignments of substrings of the read that become seeds, followed by a gap-aware dynamic programming-based seed extension to obtain the final alignment. The index is built using a variant of [105] and occupies 2.2 GB of disk space for the human reference.

For alignment every read and its reverse complement are divided into overlapping subsequences that become the potential seeds. Each seed is aligned to the reference using the FM index in an ungapped fashion, but allowing a configurable number of mismatches. For a given read, if the FM index search fails to match the seed to the reference at some location i it backtracks and attempts a base substitution at that position that can be successfully matched and in a way that maximizes the overall alignment quality. It performs up to k (configurable, default 1) such substitutions per read. Since the substitutions are made in a greedy manner the alignment result may not be globally optimal. To avoid potentially excessive backtracking Bowtie builds both a forward and a mirror index of the reference genome and attempts to match both ends of the read one after the other, with the forward and mirror index respectively attempting to obtain a high scoring alignment of both ends. The number of backtracking steps is further cut off at a hard threshold for performance reasons.

The FM index search produces sets of ranges between which each seed matches. Seeds are assigned priorities based on the number of potential matches, where seeds that have fewer potential matches (i.e. map more uniquely) are given higher priority. Seeds are then selected in priority order for exact resolution. Using the BWT LF property and the FM occurrence array the selected range is resolved to an actual matching location for a given seed. The seeds are then extended in both directions using an adaptation of SIMD-accelerated Smith Waterman alignment via dynamic programming. The striped Smith-Waterman algorithm[67] that Bowtie 2's alignment is based on allows end-to-end alignment in the neighbourhood of the seed with arbitrary numbers of mismatches and gaps and is base quality aware. When reads align to repetitive regions there may be many thousands of match locations for the seeds. To more accurately position these reads Bowtie employs a reseeding strategy where the seeds are selected from a read multiple times using a sliding window in an attempt to find a more accurate seed location.

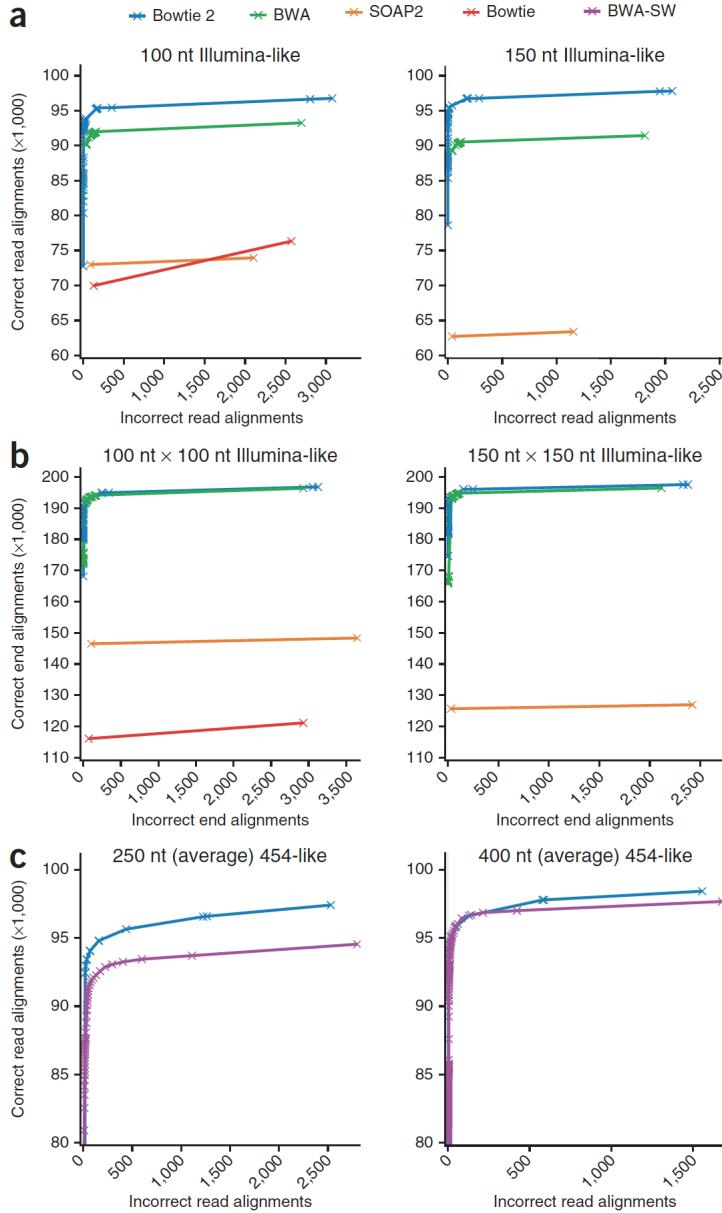


Figure 2.15: Performance comparison of Bowtie 2 to other callers on simulated data
a) 100 and 150 bp unpaired, b) 100x100 and 150x150 paired, and c) 250 and 400 long reads.[112]

Bowtie uses information about paired-end reads to attempt to perform a more accurate alignment of the pairs. When the user supplies an expected DNA fragment length and read orientation and one read in a pair is fully aligned, Bowtie calculates a window of reference coordinates where the other read in the pair would be expected to reside and attempts to align the other read in that window using dynamic programming. If it's not able to produce a high quality alignment in this fashion it will attempt to align the other read using the full normal procedure.

A performance comparison by the tool's author Ben Langmead from [112] is shown in Figure 2.15. Bowtie 2 can be seen to be the same or slightly better than BWA and significantly better than Bowtie and SOAP2 on the simulated dataset.

BWA BWA is a tool by Heng Li that implements a series of algorithms for genome alignment based on the BWT and the FM index and described in a series of papers - [131], [130], [122]. Although the original BWA algorithm was focused towards reads that are fewer than 50 basepairs long and is no longer relevant, the subsequent BWA-SW, and BWA-MEM algorithms remain very popular. Since BWA-MEM is seen as the ultimate successor to the previous BWA algorithms we focus our discussion on this implementation of alignment. The following common notation is adopted (see Table ??)

Table 2.6: BWA Common notation[124],[122]

Symbol	Description
$\Sigma = \{\$\text{, } A, C, G, T, N\}$	alphabet of DNA strings with lexicographical order $\$ < A < C < G < T < N$
$\$$	Is a sentinel character
N	Is an ambiguous DNA base
T	String: $T = a_0a_1\dots a_{n-1}$ with $a_{n-1} = \$$
$ T $	Length of T including sentinels: $ T = n$
$T[i]$	The i -th symbol in string T : $T[i] = a_i$
$T[i, j]$	Substring: $T[i, j] = a_i \dots a_j$
T_i	Suffix: $T_i = T[i, n-1]$
S	Suffix array; $S(i)$ is the position of the i -th smallest suffix
B	BWT: $B[i] = T[S(i)-1]$ if $S(i) > 0$ or $B[i] = \$$ otherwise
$C(a)$	Accumul. count array: $C(a) = \{0 \leq i \leq n-1 : T[i] < a\} $
$O(a, i)$	Occurrence array: $O(a, i) = \{0 \leq j \leq i : B[j] = a\} $
$P \circ W$	String concatenation of string P and W
Pa	String concatenation of string P and symbol a : $Pa = P \circ a$
\bar{P}	Watson-Crick reverse complement of DNA string P

The *suffix array interval* $l^l(P), l^u(P)$ of a string P is defined as:

$$\begin{aligned} l^l(P) &= \min \{k : P \text{ is the prefix of } T_{S(k)}\} \\ l^u(P) &= \max \{k : P \text{ is the prefix of } T_{S(k)}\} \\ l^s(P) &= l^u(P) - l^l(P) + 1 - \text{the interval size} \end{aligned}$$

Based on the definition of the FM index:

$$I^l(aP) = C(a) + O(a, I^l(P) - 1) \quad (2.2)$$

$$I^u(aP) = C(a) + O(a, I^u(P)) - 1 \quad (2.3)$$

and $I^l(aP) \leq I^u(aP)$ if and only if aP is a substring of T .

A string that terminates with a $\$$ is called a *text*. A text may have multiple sentinels and every sentinel has a different lexicographical rank i.e. given a text T if there exist $T[i] = \$$ and $T[j] = \$$, then $T[i] < T[j]$ iff $i < j$. Given an ordered set of texts their ordered string concatenation is called a *collection*. Given a set of DNA texts R_0, \dots, R_n , let $T = R_0\overline{R_0}R_1\overline{R_1}, \dots, R_{n-1}\overline{R_{n-1}}$ be a bidirectional collection of R . The FM index of T is called the FMD index, and the *bi-interval* of P is $[l^l(P), l^l(\overline{P}), l^s(P)]$. Using the fact that we can compute the suffix array interval of aP via Equation 2.2, and that $I^s(c\overline{P}) = I^s(cP)$ the full bi-interval of aP can be derived. This can be used, as in Algorithms 2 and 1 to bi-directionally extend a substring match P and its complement \overline{P} in either direction.

Algorithm 1: Backward extension[124]

Input: Bi-interval $[k, l, s]$ of string W and a symbol a
Output: Bi-interval of string aW

Function BACKWARDEXT($[k, l, s], a$) **begin**

for $b \leftarrow 0$ to 5 do	$k_b \leftarrow C(b) + O(b, k - 1)$ $s_b \leftarrow O(b, k + s - 1) - O(b, k - 1)$
$l_0 \leftarrow l$	
$l_4 \leftarrow l_0 + s_0$	
for $b \leftarrow 3$ to 1 do	$l_b \leftarrow l_{b+1} + s_{b+1}$
$l_5 \leftarrow l_1 + s_1$	
return $[k_a, l_a, s_a]$	

Algorithm 2: Forward extension[124]

Input: Bi-interval $[k, l, s]$ of string W and a symbol a
Output: Bi-interval of string Wa

Function FORWARDEXT($[k, l, s], a$) **begin**

$[l', k', s'] \leftarrow$ BACKWARDEXT($[l, k, s], \bar{a}$)	
return $[k', l', s']$	

A MEM (*maximal exact match*) is an exact pattern match to the index that cannot be extended in either direction, and a SMEM (*supermaximal exact match*) is a MEM that is not contained in other MEMs of the pattern. Building on match extension, Algorithm 3[124] provides a mechanism for finding SMEMs of a read in the reference sequence. BWA-MEM generates SMEMs that cover each position in a read, it then performs greedy SMEM filtering and chaining, linking together SMEMs that are located close to each other and filtering out chains that are contained in other longer chains. The list of seeds generated through this process is ranked by the length of its chain and the length of the seed itself.

Algorithm 3: Finding super-maximal exact matches (SMEMs)[124]

Input: String P and start position i_0 ; $P[-1] = 0$
Output: Set of bi-intervals of SMEMs overlapping i_0

Function SUPERMEM1(P, x) **begin**

- Initialize Curr, Prev and Match as empty arrays*
- $[k, l, s] \leftarrow [C(P[i_0]), C(\overline{P[i_0]}), C(P[i_0] + 1) - C(P[i_0])]$
- for** $i \leftarrow i_0 + 1$ **to** $|P|$ **do**
 - if** $i = |P|$ **then**
 - Append $[k, l, s]$ to Curr
 - else**
 - $[k', l', s'] \leftarrow \text{FORWARDEXT}([k, l, s], P[i])$
 - if** $s' \neq s$ **then**
 - Append $[k, l, s]$ to Curr
 - if** $s' = 0$ **then**
 - break
 - $[k, l, s] \leftarrow [k', l', s']$
- Swap array Curr and Prev*
- $i' \leftarrow |P|$
- for** $i \leftarrow i_0 - 1$ **to** -1 **do**
 - Reset Curr to empty
 - $s'' \leftarrow -1$
 - for** $[k, l, s]$ **in** Prev **do**
 - $[k', l', s'] \leftarrow \text{BACKWARDEXT}([k, l, s], P[i])$
 - if** $s' = 0$ **or** $i = -1$ **then**
 - if** Curr is empty **and** $i + 1 < i' + 1$ **then**
 - $i' \leftarrow i$
 - Append $[k, l, s]$ to Match
 - if** $s' \neq 0$ **and** $s' \neq s''$ **then**
 - $s'' \leftarrow s'$
 - Append $[k, l, s]$ to Curr
 - if** Curr is empty **then**
 - break
 - Swap Curr and Prev*

- return** Match

The seeds are extended using a banded affine gap-penalty dynamic programming alignment implementation. A number of heuristics is deployed to limit the search space. For paired-end mapping, BWA-MEM performs Smith-Waterman alignment in a window of $[\mu - 4\sigma, \mu + 4\sigma]$ from a mapped read, when its mate is unmapped. When selecting how to match paired alignments BWA-MEM uses a score of $S_{i,j} = S_i + S_j - \min \{-a \log_4 P(d_{i,j}, U)\}$, where S_i and S_j are individual read alignment scores, $d_{i,j}$ is the insert distance between two reads, $P(d_{i,j})$ is the probability of observing $d_{i,j}$ under a normal model of insert sizes, and U is a sensitivity threshold. Figure 2.16 shows a comparison performed by BWA-MEM author between BWA-MEM and other popular aligners. BWA-MEM is seen to outperform all aligners except the commercial tool Novoalign on accuracy and be in the top two tools for speed.

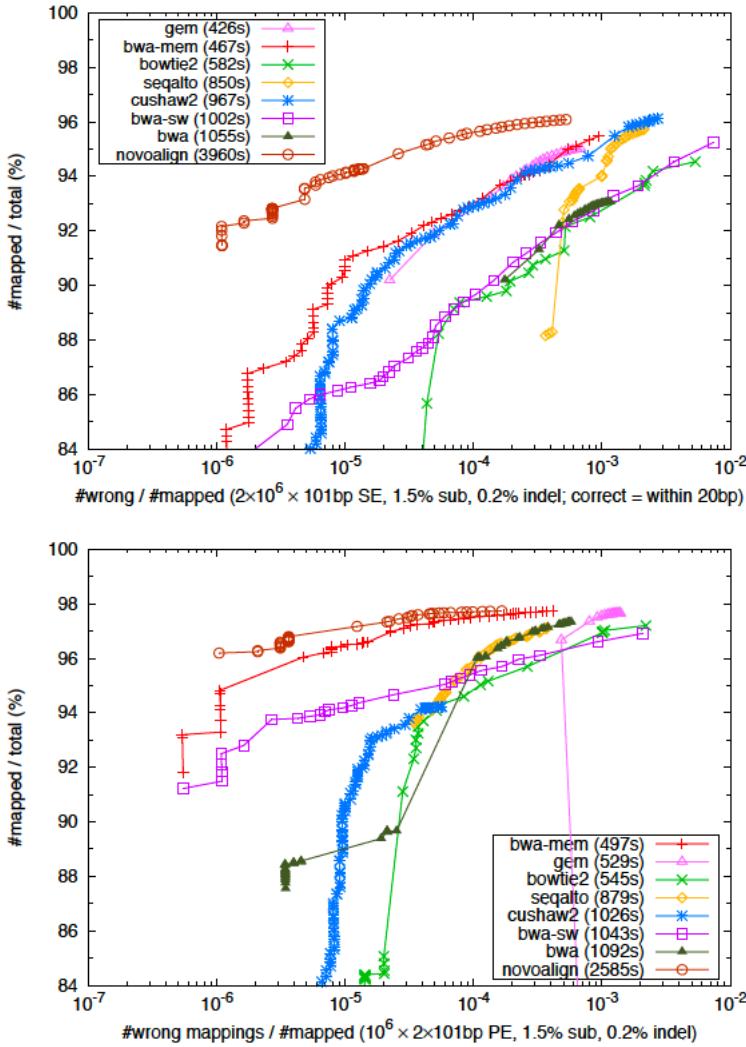


Figure 2.16: Performance of multiple aligners on simulated single-end and paired-end 101 bp reads.[122]

Minimap2 Bowtie 2 and BWA-MEM that have been presented so far represent state-of-the-art algorithms for the alignment of short (<500 bp) reads to a reference genome using an FM-index. There are, however, several legitimate cases for the alignment of much longer reads to one or several references, and for finding overlaps between groups of long reads. New DNA sequencing technologies, including Single Molecule Real-Time (SMRT)[179] from Pacific Biosciences Inc., and Oxford Nanopore Technologies' (ONT)[137], have been producing reads that range from 1000 bp to $> 10^6$ bp albeit with a much higher error rate than those from Illumina short reads. These long reads can be used to resolve structural variation encountered in repeat-enriched areas of the genome that are not possible to uniquely resolve with shorter Illumina-based reads[183]. Additionally, tools such as the GATK HaplotypeCaller (see Section 2.2.4) create locally assembled haplotypes (that can be thousands of basepairs long) and align these to the reference sequence to detect variants. Yet, tools that have been developed primarily for short read alignment (like BWA-MEM) either crash or perform exceedingly slow when aligning long read

sequences. Minimap 2[126] has been created to solve the efficient alignment problem for long reads using hashmap-based approach. All of the formulas in this section have been reproduced from the 2018 paper by Heng Li[126].

Minimap2 uses a hashmap approach based on building a database of representative k-mers, called minimizers, for a reference sequence, and searching against this database. Minimizers have been introduced by Roberts et al.[181] as a way of reducing the storage required for comparing biological sequences. The intuitive notion is that if one is comparing two strings with a significant overlap, extracting and comparing a set of representative k-mers from both strings will produce a match between the k-mers that can then be used as the basis for a seed-and-extend algorithm for more accurate matching. In this case, rather than storing all k-mers, one may store only the minimizers and use them when searching for seeds.

Given a string $T_i = a_1a_2\dots a_n$ where $a_i \in \Sigma = \{A, C, G, T, N\}$, a k-mer triple is a tuple (s, i, p) that stores the k-mer string s , index i that identifies T_i and p the start index of s in T_i . Assume there exists an order on elements of Σ . If we consider w consecutive k-mers, covering $w + k - 1$ letters of T_i , then the smallest k-mer is called a (w, k) minimizer of T_i (See Figure 2.17). If two strings have a substring of length $w + k - 1$ in common, then they have a (w, k) minimizer in common.

Position	1	2	3	4	5	6	7	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	2	3	1	0	3	4	3	4	2	6	4	7	2	8	1	4	7	5	1
<i>k</i> -mers	2	3	1					4	2	6	4	7	2	8					
with		3	1	0					2	6	4	7	2	8	1				
minimizer			1	0	3				6	4	7	2	8	1	4				
in				0	3	4				4	7	2	8	1	4	7			
bold					3	4	3				7	2	8	1	4	7	5		
(a)								(b)			2	8	1	4	7	5	1		

Figure 2.17: Examples of minimizers - a) (5,3)-minimizer, b) (6,7)-minimizer.[181]

Minimap2 builds a list of minimizers of the reference sequence and stores them in a hash table where the key is the hash of the minimizer string and the value is the list of offsets into the reference where that minimizer is found. For each query sequence that needs to be aligned, minimap2 builds minimizers of the query and searches the reference hash table for them. The matches that are found become alignment anchors and sets of anchors that are in the same order and orientation in the query and the reference are joined into chains. The gaps between anchors in a chain are then filled using dynamic programming with a 2-piece affine gap penalty to produce a final base-level alignment.

On simulated long reads (Figure 2.7 a)), minimap2 is more accurate than other aligners and is up to 30 times faster. On real SMRT reads minimap2 was 70 times faster than short-read aligners. On short reads (Figure 2.7 b)), minimap2 was less accurate than BWA-MEM but was up to 3 times faster. Using data from a synthetic diploid cell-line[134] minimap2 shows a higher false negative rate compared to BWA-MEM (FNR 2.6% vs. 2.3%), but a lower false positives per million bases (FPPM,

7.0 vs. 8.8) for SNPs, and similar performance for indels.

2.2.3 Raw Data QC

The data that is generated as part of NGS experiments can have widely varying quality, and may suffer from systematic biases introduced by the experimental protocol employed, the technology used, as well as individual events such as sample contamination[115, 12, 56, 31]. The effect of low quality data such as sequencing errors on downstream analysis can be quite significant, not only because it may introduce false positive variants in the final output, but also because it can produce a large number of potentially variant sites that need to be evaluated and will consume a large amount additional computational resources to eventually rule out. A wide variety of computational methods exist for examining the data after it has been generated in order to identify and fix or filter out low quality data. The most widely used tools include Picard[173], FastQC[16], QC Toolkit[170], QC-Chain[243], and FASTX-Toolkit[81]. These tools evaluate data at two granularities - read-level, and sample-level. Some of these can act on sequencing data pre-alignment, while others either work post-alignment or actively make use of a built-in aligner. The QC process is generally organized into a QC pipeline (see Figure 2.18). Furthermore, some tools, like FastQC and Picard tend to only collect various QC metrics and generate reports, leaving the filtering based on these metrics up to the user, while other tools, such as QC Toolkit and QC-Chain perform the actual filtering themselves.

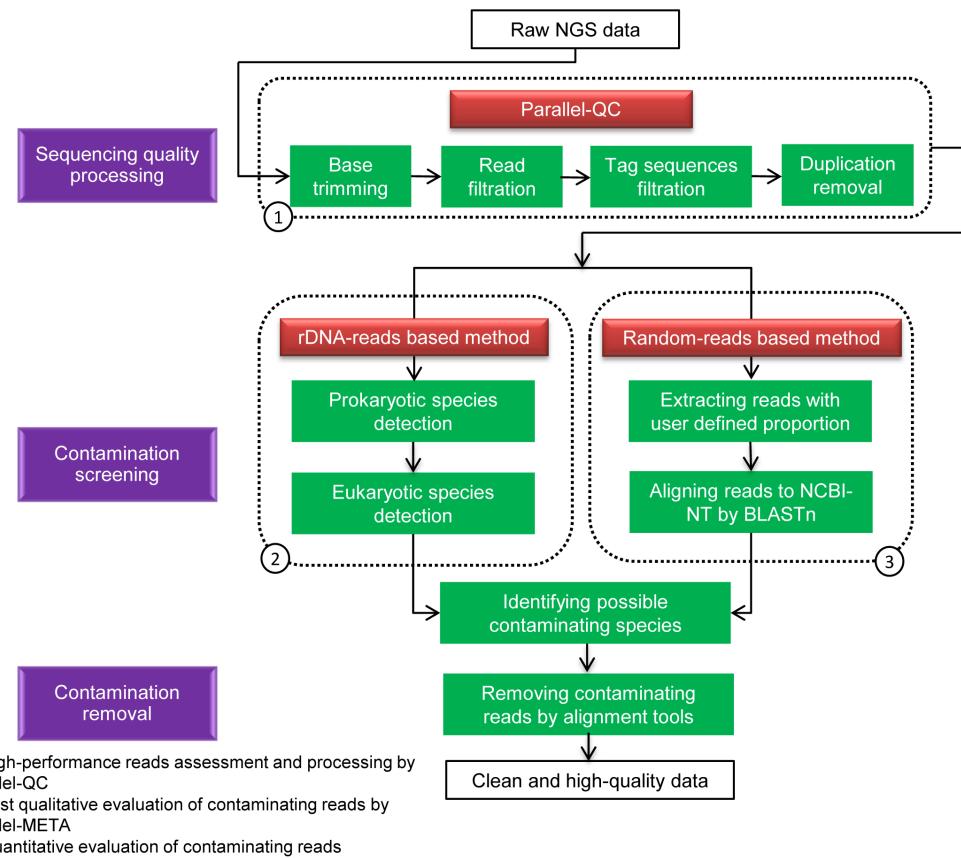


Figure 2.18: The QC-Chain pipeline involves trimming low quality bases, removal of adapter sequences, removal of duplicate reads, and contamination detection.[243].

At individual read level the following QC measures are of interest:

Base Quality Distribution - Reads in FASTQ format have a base-quality score associated with each nucleotide which serve as estimates of the probability that the base has been called correctly by the base-caller software. Bases towards the end of Illumina reads tend to suffer from deteriorating quality([56] and may not be usable for downstream analysis.

Adapter Sequence Presence - The NGS library preparation protocol involves ligating an adapter sequence to the end of DNA fragments in order to bind the fragment to the sequencing flowcell. Sometimes the sequencing process does not stop at the end of the actual DNA molecule and also sequences the adapter. It is necessary to detect and trim out these adapter sequences as they do not belong to the genome[23].

Duplicate Detection - in NGS libraries that use PCR amplification a particular DNA fragment may be sequenced multiple times resulting in increased apparent, and depleted actual coverage of that region of the genome, possibly influencing downstream variant calling efforts. Both Picard[173] and Samtools[121] have utilities for detecting PCR duplicates, although the additional benefit of this processing step may be marginal[58].

Sample Contamination/Sample Swap - Sample contamination during library preparation may result in the presence of foreign DNA in the generated sequence. The foreign sequence may originate from the same or from foreign species. Furthermore, entire samples may be swapped or mislabelled (for instance, cancer samples with tumour from one patient, but normal sample from another patient). Contamination may be detected by aligning reads to a panel of potentially contaminating species' reference genomes[243], or when a surprising number of variants is found in a given genome.

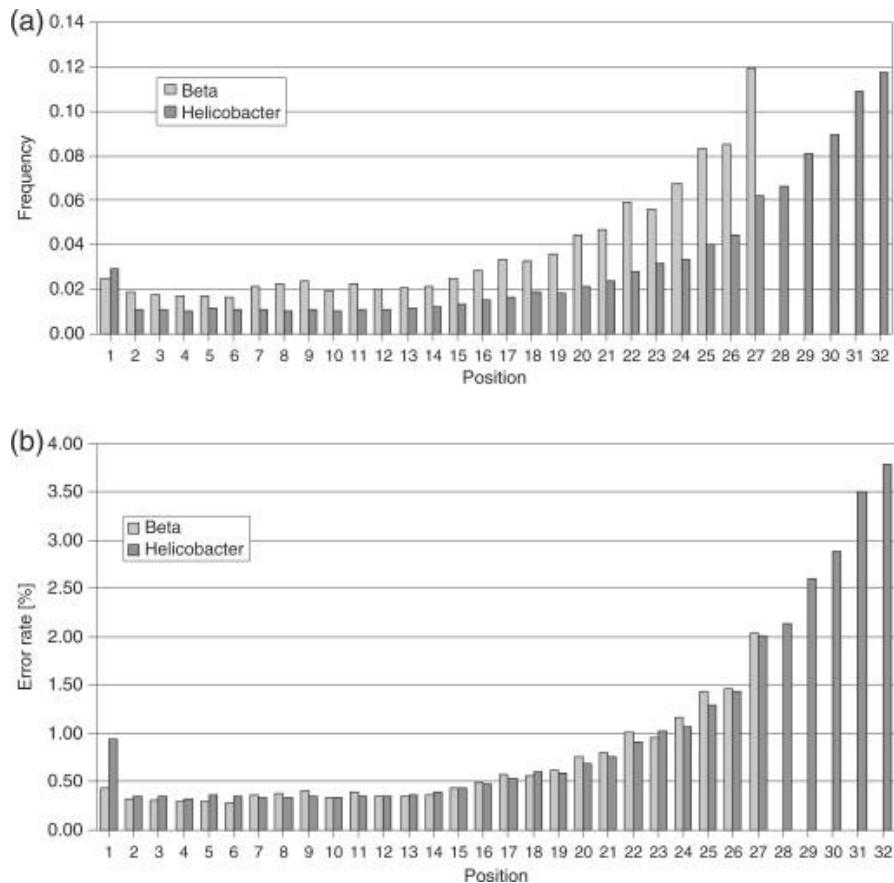


Figure 2.19: Frequency of wrong base calls in Illumina reads based on position in the read from 5' to 3'.[56].

At the sample level the following QC measures are important:

Insert Distribution - summary statistics (mean, median, standard deviation) related to the distribution of the distance between paired-end reads.

Per-base Quality Distribution - distribution of base qualities for each position in a read, aggregated over all reads.

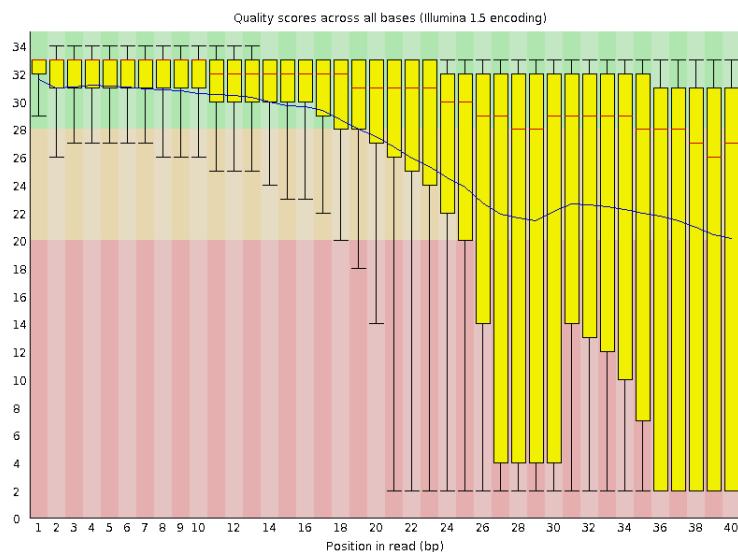


Figure 2.20: from <https://www.bioinformatics.babraham.ac.uk/projects/fastqc>.

Per-read Quality Distribution - distribution of average read qualities.

Per-base Sequence Content Distribution - distribution of nucleotide frequencies for each position in a read, aggregated over all reads.

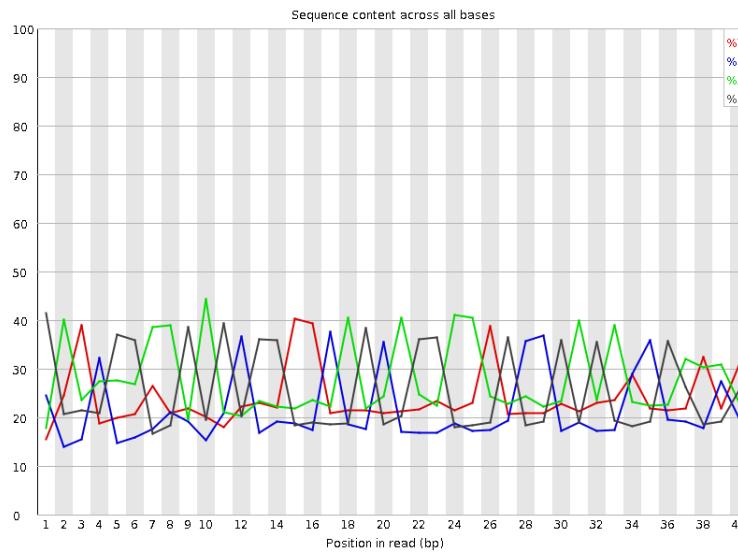


Figure 2.21: from <https://www.bioinformatics.babraham.ac.uk/projects/fastqc>.

Per-read GC Content Distribution - distribution of average GC content per read.

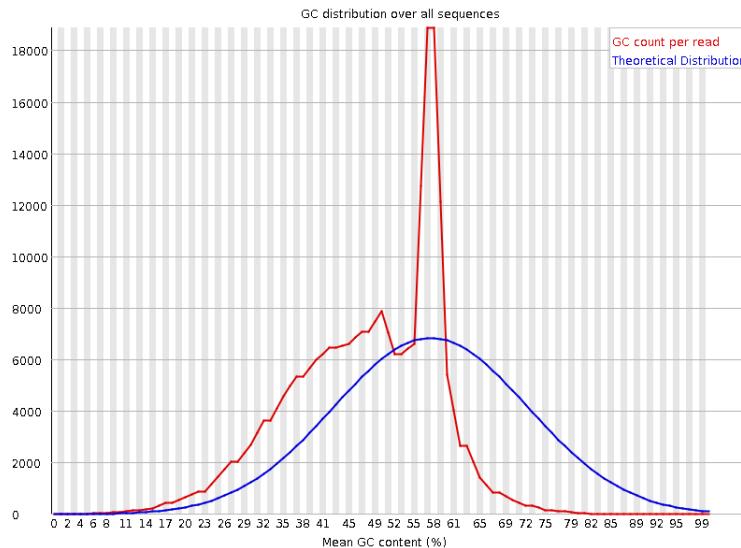


Figure 2.22: from <https://www.bioinformatics.babraham.ac.uk/projects/fastqc>.

Read-length Distribution - summary statistics of read lengths.

Per-base N Content - distribution of uncalled bases for each position in a read, aggregated over all reads.

Sequence Duplication Distribution - distribution of counts for duplicated sequences.

Overrepresented Sequence Distribution - frequency of sequence fragments that occur more frequently than expected.

Per-base Adapter Content Distribution - frequency of adapter sequence presence for each position in a read, aggregated over all reads.

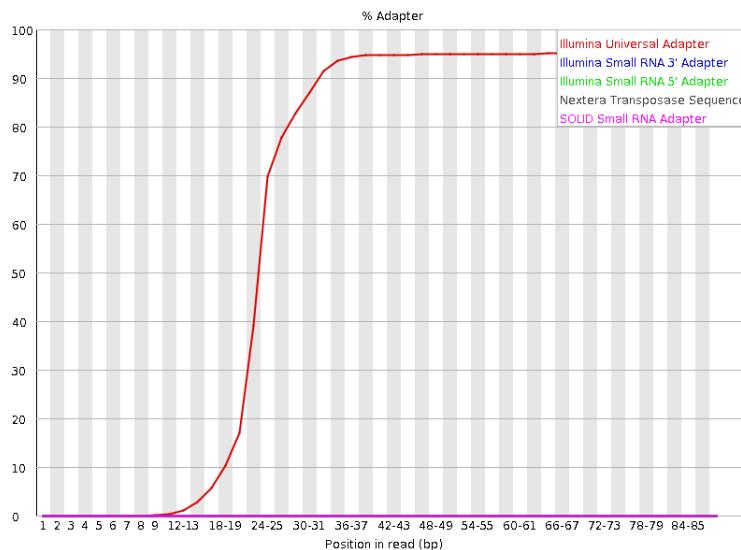


Figure 2.23: from <https://www.bioinformatics.babraham.ac.uk/projects/fastqc>.

Many other QC metrics can be of interest and are listed on the individual tools' pages (for instance <https://broadinstitute.github.io/picard/picard-metric-definitions.html>). Once collected, these metrics can be used for visualization, manual curation, threshold-based filtering, or machine learning approaches to QC.

2.2.4 Germline SNP Calling

Single Nucleotide Polymorphisms or SNPs are locations in an individual's genome where that individual differs from the reference sequence at a single position. The reference sequence is haploid i.e. it provides a single base (for instance T) at every genomic location, whereas the human genome is diploid (there are two copies of each chromosome, and thus two bases at each location) for chromosomes 1-22, and chromosome X for females, while being haploid for chromosomes X and Y for males. Thus, at each genomic location, the human genome may be:

Homozygous Reference - When both alleles carried by the individual at that location match the reference.

Heterozygous - When one allele matches the reference and one is different from the reference.

Homozygous Alternate - When both alleles are the same and different from the reference.

Multiallelic - When both alleles are different from the reference and are different from each other[93].

SNPs are the most common type of genomic variant, with every individual carrying over 3 million SNPs on average[197]. Furthermore, the presence of certain SNPs is strongly associated with disease[41], where some SNPs are known to be causative[100], while others, are merely associated with a disease phenotype[190]. A large number of scientific studies[92] and clinical practice[236] is thus enabled by efficient and comprehensive characterization of the gamut of human SNPs to assess their contribution to disease risk, see Figure 2.24.

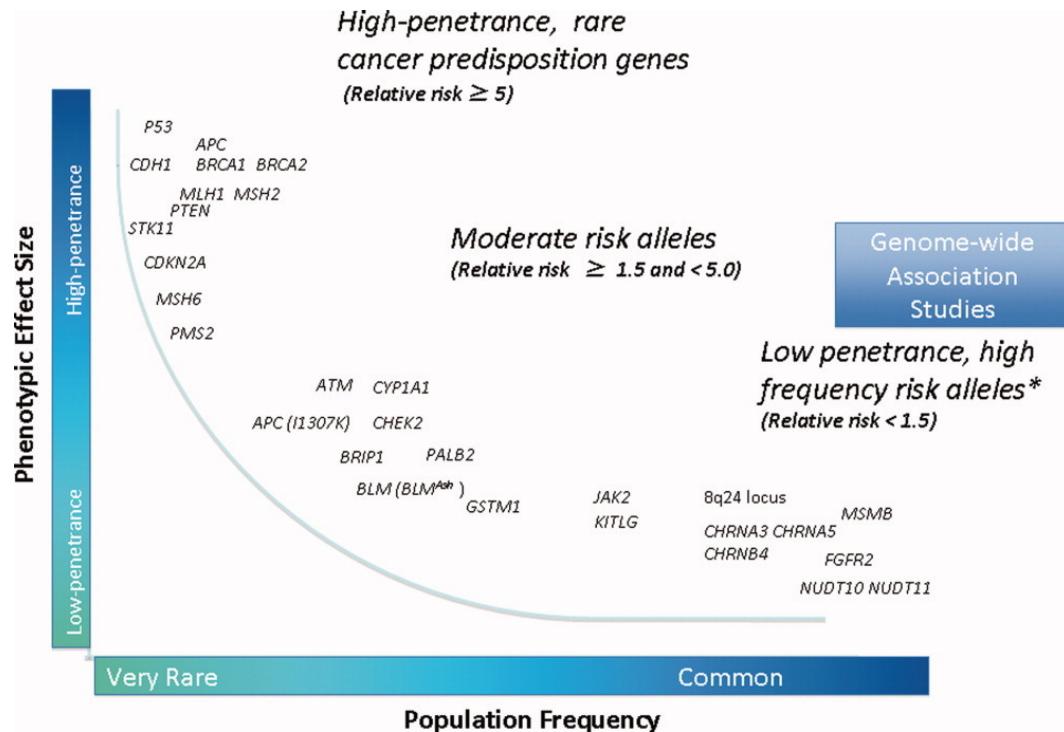


Figure 2.24: Distribution of mutations by population frequency against phenotypic effect size.[228].

There are a number of methods that have been used for assessing SNPs with the aid of microarray technology[90], but here we focus on methods that make use of Next Generation Sequencing (NGS). Since the primary data type generated by NGS is a sequencing read, most presently used methods for SNP detection rely on investigating the collection of sequencing reads that overlap each genomic locus and comparing the observed data to the reference sequence. It is important to distinguish two typically separate activities that take place as part of SNP calling - variant calling, and genotyping. Variant calling attempts to locate positions in the sample genome where that sample is different from the reference, whereas genotyping attempts to assign an actual genotype (e.g. homozygous-alternate), along with a measure of confidence, to each putative variant. We present several of the key computational methods currently used in SNP calling with additional detail. These are:

- samtools
- freebayes
- GATK
- platypus

These tools have been selected because they have been developed independently, at different institutions, and have been repeatedly demonstrated to produce consistent and high-quality results. See Figure 2.25 for a recent comparison.

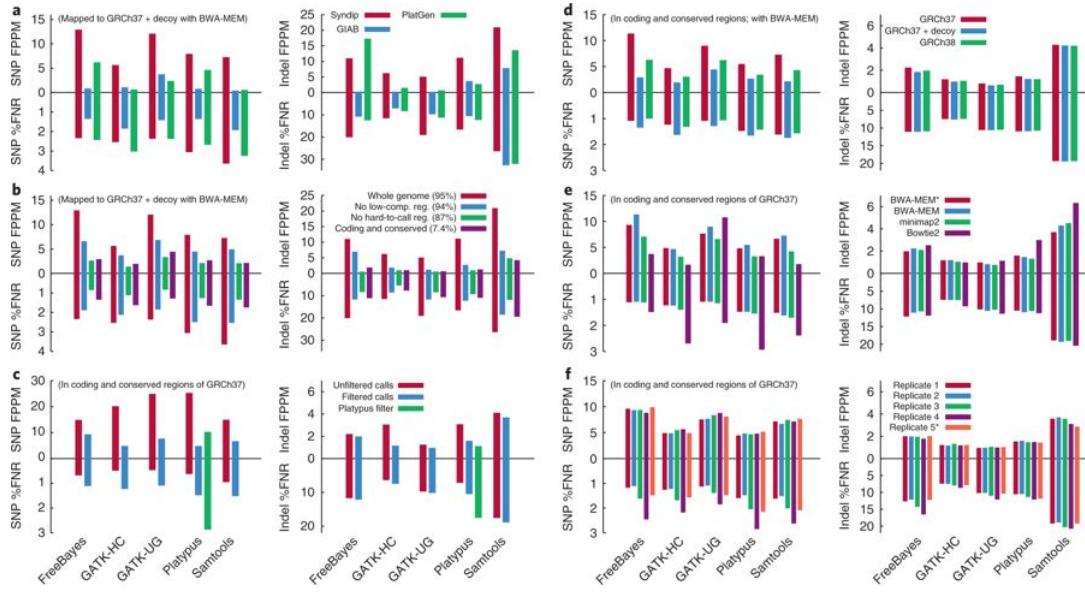


Figure 2.25: Comparison of samtools, freebayes, GATK, and platypus on three benchmark data sets - Syndip, GIAB, and PlatGen. Here FPPM - number of false positives per megabase of sequence, and FNR - false negative rate = $100 \times FN / (TP + FN)$.[134].

samtools

Samtools[135] is a software package for genomic data processing developed by Heng Li et al. in the context of the 1000 Genomes Project[37] and implemented as a C program with a CLI interface. This tool has enjoyed continued and widespread use in the bioinformatics community for the purposes of small variant calling (including SNPs). All of the mathematical results in this section are reproduced from the 2011 paper by Li[121] that describes the method, as well as a set of mathematical notes made available separately by Li[125] in 2010.

Although the samtools framework could be extended to support calling multi-allelic sites, the framework, as-published, has been developed for calling only bi-allelic variants. Table 2.7 contains commonly used definitions.

It is additionally assumed that there are n individuals being sequenced with the i -th individual having ploidy m_i (typically 2 in practice). At a particular genomic locus, the sequence read data for the i -th individual is d_i and the genotype is g_i , an integer in $[0, m_i]$, counting the number of reference alleles in the individual at that locus. Furthermore, it is assumed for simplicity that data at individual genomic loci are independent (which isn't necessarily true), as are sequencing and mapping errors between loci and individuals.

Because of the above independence assumptions the joint likelihood function of the data observed for all individuals factors as a product of individual likelihood

Table 2.7: Samtools common definitions

Symbol	Description
n	Number of samples
m_i	Ploidy of the i -th sample ($1 \leq i \leq n$)
M	Total number of chromosomes in samples: $M = \sum_i m_i$
d_i	Sequencing data (bases and qualities) for the i -th sample
g_i	Genotype (the number of reference alleles) of the i -th sample ($0 \leq g_i \leq m_i$) ¹
ϕ_k	Probability of observing k reference alleles ($\sum_{k=0}^M \phi_k = 1$)
$P(A)$	Probability of an event A
$\mathcal{L}_i(\theta)$	Likelihood function for the i -th sample: $\mathcal{L}_i(\theta) = P(d_i \theta)$

functions:

$$\mathcal{L}(\theta) = \prod_{i=1}^n \mathcal{L}_i(\theta) \quad (2.4)$$

Suppose that a single sample i represents an individual of ploidy m_i and a given locus is covered by k reads. The sequencing data d_i is composed of an array of bases where each element has value 1 representing the reference allele and is 0 otherwise.

$$d_i = (b_1, \dots, b_k) = (\underbrace{1, \dots, 1}_l, \underbrace{0, \dots, 0}_{k-l})$$

The error probability of the j -th base is ϵ_j , which is taken to be the larger between sequencing and mapping errors for that read. Under the independence assumptions above:

$$\mathcal{L}_i(0) = P(d_i|0) = \prod_{j=1}^l \epsilon_j \prod_{j=l+1}^k (1 - \epsilon_j) = \left(1 - \sum_{j=l+1}^k \epsilon_j + o(\epsilon^2) \right) \prod_{j=1}^l \epsilon_j \quad (2.5)$$

$$\mathcal{L}_i(m_i) = P(d_i|m_i) = \left(1 - \sum_{j=1}^l \epsilon_j + o(\epsilon^2) \right) \prod_{j=l+1}^k \epsilon_j \quad (2.6)$$

For $0 < g_i < m_i$:

$$\begin{aligned}
\mathcal{L}_i(g_i) = P(d_i|g_i) &= \sum_{a_1=0}^1 \cdots \sum_{a_k=0}^1 \Pr\{d_i|B_1 = a_1, \dots, B_k = a_k\} \Pr\{B_1 = a_1, \dots, B_k = a_k|g\} \\
&= \sum_{\vec{a}} \left(\frac{g}{m}\right)^{\sum_j a_j} \left(1 - \frac{g}{m}\right)^{k - \sum_j a_j} \cdot \prod_j p_j(a_j) \\
&= \left(1 - \frac{g}{m}\right)^k \prod_j \sum_{a=0}^1 p_j(a) \left(\frac{g}{m-g}\right)^a \\
&= \left(1 - \frac{g}{m}\right)^k \prod_{j=1}^l \left(\epsilon_j + \frac{g}{m-g}(1 - \epsilon_j)\right) \prod_{j=l+1}^k \left(1 - \epsilon_j + \frac{\epsilon_j g}{m-g}\right) \\
&= \left(1 - \frac{g}{m}\right)^k \left\{ \left(\frac{g}{m-g}\right)^l + \left(1 - \frac{g}{m-g}\right) \left(\sum_{j=1}^l \epsilon_j - \sum_{j=l+1}^k \epsilon_j \right) + o(\epsilon^2) \right\}
\end{aligned}$$

where $a = \sum_j a_j$ and

$$p_j(a) = \begin{cases} \epsilon_j & a = 1 \\ 1 - \epsilon_j & a = 0 \end{cases}$$

In particular, for a diploid sample ($m = 2$), the likelihoods for $g = 0, 1, 2$ are

$$\mathcal{L}(0) = \prod_{j=1}^l \epsilon_j \prod_{j=l+1}^k (1 - \epsilon_j) \quad (2.7)$$

$$\mathcal{L}(1) = \frac{1}{2^k} \quad (2.8)$$

$$\mathcal{L}(2) = \prod_{j=1}^l (1 - \epsilon_j) \prod_{j=l+1}^k \epsilon_j \quad (2.9)$$

For instance, taking $g_i = 2$ (i.e. the true genotype is homozygous-reference) as an example, and under above independence assumptions, the likelihood of observing the data d_i is the likelihood of sampling l reads without error (the reads match the reference) and $k - l$ reads with error (the reads do not match the reference).

Let $\Phi = \{\phi_k\}$ for $k \in [0, m_i]$ be a prior distribution of genotype probabilities (a model from population genetics, such as Wright-Fisher, can be used, or an empirical distribution from another study), the actual genotype for individual i at the given locus is estimated via Bayes' Rule as:

$$\hat{g}_i = \operatorname{argmax}_{g_i} \Pr\{G_i = g_i|d_i, \Phi\} = \operatorname{argmax}_{g_i} \frac{P(d_i|g_i)\phi_k}{\sum_{h_i} P(d_i|h_i)\phi_h} = \operatorname{argmax}_{g_i} \frac{\mathcal{L}(g_i)\phi_k}{\sum_{h_i} \mathcal{L}(h_i)\phi_h} \quad (2.10)$$

The variant quality is defined as:

$$Q_{\text{var}} = -10 \log_{10} \Pr\{G = m_i|d_i, \Phi\} \quad (2.11)$$

i.e. the Phred-scaled probability that the locus is homozygous reference given the observed data. The locus is called *variant* if the variant quality exceeds a certain pre-defined threshold.

Equations 2.10 and 2.11 thus represent the key computations corresponding to the activities of SNP genotyping and variant calling that are performed by samtools for each genomic locus in a sample.

freebayes

freebayes[78] is a C software package implemented by E. Garrison for discovery and genotyping of SNPs, indels, and other small variants that builds on a previous method by G. Marth[146]. Where small means that the variant length is smaller than the size of a sequencing read. Unlike samtools, which looks at all genomic loci independently, freebayes uses local sequence context to guide detection and genotyping of variants, additionally freebayes builds variant haplotypes[48] i.e. groups of variants that are inherited together on the same DNA molecule. All of the figures and mathematical formulas in this section are reproduced from the 2012 preprint by Garrison et al[78].

Table 2.8: Freebayes common definitions

Symbol	Description
n	Number of samples
m_i	Copy number of the i -th sample ($1 \leq i \leq n$)
M	Total number of copies of each locus in all samples: $M = \sum_i m_i$
$\{b_j : j \in [1, K]\}$	Set of K distinct alleles present among M copies
$\{C_j : j \in [1, K]\}$	Corresponding set of allele counts for each b_K
$\{f_j : j \in [1, K]\}$	Corresponding set of allele frequencies for each b_K
G_i	Unphased genotype of individual i
$\{b_{ij} : i \in [1, n], j \in [1, k_i]\}$	Set of k_i distinct alleles of G_i
$\{c_{ij} : i \in [1, n], j \in [1, k_i]\}$	Set of k_i allele counts for each b_{ij}
$\{f_{ij} : i \in [1, n], j \in [1, k_i], f_i = c_i/k_i\}$	Set of allele frequencies of G_i
$B_i : B_i = m_i$	Multiset of alleles equivalent to G_i
$R_i = \{r_j : j \in [1, s_i]\}$	Set of s_i sequencing observations for each sample i s.t. there are $\sum_{i=1}^n R_i $ reads at a given locus
$\{q_j : j \in [1, s_i]\}$	Mapping quality, i.e. probability that read r_j is mis-mapped against the reference

For a set of n individuals the conditional probability of a combination of genotypes

given the observed data is assessed simultaneously as:

$$P(G_1, \dots, G_n | R_1, \dots, R_n) = \frac{P(G_1, \dots, G_n) P(R_1, \dots, R_n | G_1, \dots, G_n)}{P(R_1, \dots, R_n)} \quad (2.12)$$

$$P(G_1, \dots, G_n | R_1, \dots, R_n) = \frac{P(G_1, \dots, G_n) \prod_{i=1}^n P(R_i | G_i)}{\sum_{\forall G_1, \dots, G_n} P(G_1, \dots, G_n) \prod_{i=1}^n P(R_i | G_i)} \quad (2.13)$$

For a single sample at a particular locus there are R_i reads, and k_i observed alleles - $B'_i = b'_1, \dots, b'_{k_i}$, which correspond to b_1, \dots, b_i underlying alleles represented at the given locus. For each observed allele b'_i there is a corresponding count of observations o_f s.t. : $\sum_{j=1}^{k_i} o_j = s_i$ and each b'_i corresponds to a true allele b_i . The probability of a single observation b'_i given a genotype in a single sample is:

$$P(b'_i | G) = \sum_{\forall (b_i \in G)} f_i P(b'_i | b_i) \quad (2.14)$$

where f_i is the allele frequency of b_i in G . The authors introduce the following approximation:

$$P(b' | b) = \begin{cases} 1 & \text{if } b' = b \\ P(\text{error}) & \text{if } b' \neq b \end{cases} \quad (2.15)$$

where $P(\text{error})$ is derived from the base quality score from the sequencing read. Using the above approximation the probability of observing the data R_i given the genotype is estimated as:

$$P(R_i | G) \approx \binom{s_i}{o_1, \dots, o_{k_i}} \prod_{j=1}^{k_i} f_{i_j}^{o_j} \prod_{l=1}^{s_i} P(b'_l | b_l) \quad (2.16)$$

In order to evaluate the posterior probability of a particular combination of genotypes given the data, the authors derive:

$$P(G_1, \dots, G_n | f_1, \dots, f_k) = \binom{M}{c_1, \dots, c_k}^{-1} \prod_{i=1}^n \binom{m_i}{c_{i_1}, \dots, c_{i_{k_i}}} \quad (2.17)$$

counting the number of ways of selecting a set of unphased genotypes G_i given the allele frequency spectrum f_k , and

$$P(f_1, \dots, f_k) = P(a_1, \dots, a_M) = \frac{M!}{\theta \prod_{z=1}^{M-1} (\theta + z)} \prod_{j=1}^M \frac{\theta^{a_j}}{j^{a_j} a_j!} \quad (2.18)$$

using Ewens' sampling formula[65], where θ is the population mutation rate and allele frequencies f_i are transformed to frequency counts $a_1, \dots, a_M : \sum_{c=1}^M ca_c = M$ where each a_f is the number of alleles in b_1, \dots, b_k whose allele count in the sample set is c .

Once variants are initially detected using Equation 2.13, the method continues to build local haplotypes grouping variants that are inherited together in dynamically-sized windows based on a distance threshold between successive variants. Each group of variants is combined into a haplotype observation H_i , with an assigned quality score that is the minimum of the supporting reads' mapping quality and the minimum base quality of the variant allele observations (bases that span the variants). The size of the window is determined via an iterative process where an initial variant is used as the seed, and all of the reads that overlap it are added. If these reads overlap any other variants, then these are added, along with any reads that overlap them, and so on, until no new variants can be added.

Once the window is determined, the method uses a gradient descent algorithm to find a MAP estimate of the genotype for each sample. It starts with the maximum likelihood estimate for each sample's genotype given the observed data, and then attempts to find a genotype assignment that has a higher posterior probability across all samples.

GATK

The GATK[54] is not actually a tool that's built solely for SNP calling, but is instead a comprehensive framework for genomic data analysis that includes tools for data pre-processing and QA, SNP and indel calling, CNV calling, and SV calling, as well as post-processing and filtering. It is implemented as a Java program and the latest version makes use of the in-memory computing engine Apache Spark for efficient computation over large data sets. Here we focus on the SNP calling aspects of the framework. There are two components in the GATK that deal with SNP calling, the UnifiedGenotyper, and the HaplotypeCaller. The UnifiedGenotyper is an older component that has been superceded by the HaplotypeCaller for all practical purposes and we focus our attention on it. All of the mathematical formulas and figures, as well as some of the descriptions, in this section have been reproduced from the 2010 manuscript by McKenna et al.[149], the 2011 manuscript by DePristo et al.[54], and the GATK website[2].

The GATK Best Practices pipeline (see Figure 2.26) is a set of best practices published by The Broad Institute that describe how to best use their software. In the context of this section we are primarily interested in the processes that occur in the middle panel of this figure that deal with processing aligned reads to call and genotype variants that will then be used for post-processing and filtering.

The HaplotypeCaller which is the tool that is primarily used for SNP calling takes a continued refinement approach, where the data is processed in the following sequential steps:

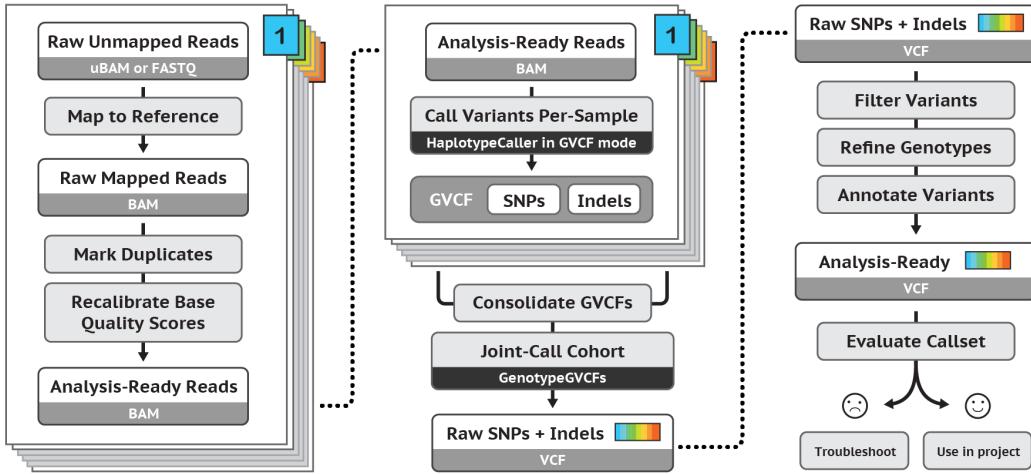


Figure 2.26: GATK Best Practices pipeline[2]

Define active regions - The program determines which regions of the genome it needs to operate on (active regions), based on the presence of evidence for variation.

Determine haplotypes by assembly of the active region - For each active region, the program builds a De Bruijn-like graph to reassemble the active region and identifies what are the possible haplotypes present in the data. The program then realigns each haplotype against the reference haplotype using the Smith-Waterman algorithm in order to identify potentially variant sites.

Determine likelihoods of the haplotypes given the read data - For each active region, the program performs a pairwise alignment of each read against each haplotype using the PairHMM algorithm. This produces a matrix of likelihoods of haplotypes given the read data. These likelihoods are then marginalized to obtain the likelihoods of alleles for each potentially variant site given the read data.

Assign sample genotypes - For each potentially variant site, the program applies Bayes' rule, using the likelihoods of alleles given the read data to calculate the likelihoods of each genotype per sample given the read data observed for that sample. The most likely genotype is then assigned to the sample.

Active regions are defined by targeting loci with high quality reads that are different from the reference and surrounded by soft-clipped bases (bases in the read that couldn't be aligned by the alignment algorithm). The region is then smoothed by a Gaussian kernel and passed onto variant calling if its profile score exceeds a pre-defined threshold.

For each active region the GATK performs local assembly by first building a de-Bruijn graph[36] using just the reference sequence for the active region. The graph is enhanced by comparing each read from the active region and creating nodes in the graph where the read differs from the graph. Edge weights between pairs of nodes are increased when a read passes through that edge. The resulting graph must

meet complexity requirements based on the ratio of non-unique to unique kmers and presence of cycles (that are a sign of repetitive sequence). When the graph does not meet complexity requirements it is automatically rebuilt with successively increased kmer sizes. If a sufficiently complex graph cannot be built, the region is discarded because the method is not able to produce a high quality variant call in this case. Once the graph is built it is refined by pruning out paths that are not supported by a sufficient number of reads. By traversing the paths in the graph and selecting those with a high aggregate weight the algorithm builds a list of possible local haplotypes within the active region. Each candidate haplotype is then realigned to the reference sequence using Smith-Waterman[202] alignment to produce a list of potential variant sites that includes SNPs and other potential variants.

Reads from the active region are aligned to each haplotype using a Hidden Markov Model via the PairHMM algorithm[57] to produce a likelihood of each read given a haplotype. The set of haplotype likelihoods is transformed to a set of likelihoods per allele where for each allele at a given site the highest scoring likelihood for a given read and a given haplotype that supports that allele is selected. Under a read independence assumption the total likelihood of the allele is computed as the product of read likelihoods.

For each variant locus the probability of each genotype is calculated using:

$$P(G|D) = \frac{P(G)P(D|G)}{\sum_i P(G_i)P(D|G_i)} \quad (2.19)$$

where D is the set of reads, G_i is the set of possible genotypes, and under the assumptions of a diploid sample, independent read observations, and independent errors. The prior probability of genotypes $P(G)$ is assumed to be uniform.

Based on the diploid and independence assumptions:

$$P(D|G) = \prod_j \left(\frac{P(D_j|H_1)}{2} + \frac{P(D_j|H_2)}{2} \right) \quad (2.20)$$

where $P(D_j|H_n)$ is the likelihood of read given the haplotype previously obtained.

The genotype with the highest $P(G|D)$ is emitted as the genotype for that variant.

Platypus

Platypus[180] is a Python and C package by Rimmer et al., that provides SNP, indel, and other small and medium (upt to 1kb) variant calling capabilities utilizing a Bayesian statistical framework built on top of reference-free local assembly of haplotypes. All of the figures and methematical formuls in this section are reproduced from the 2014 Nature Genetics manuscript by Rimmer et al. See Figure 2.27 for the general sequence of data processing steps in platypus.

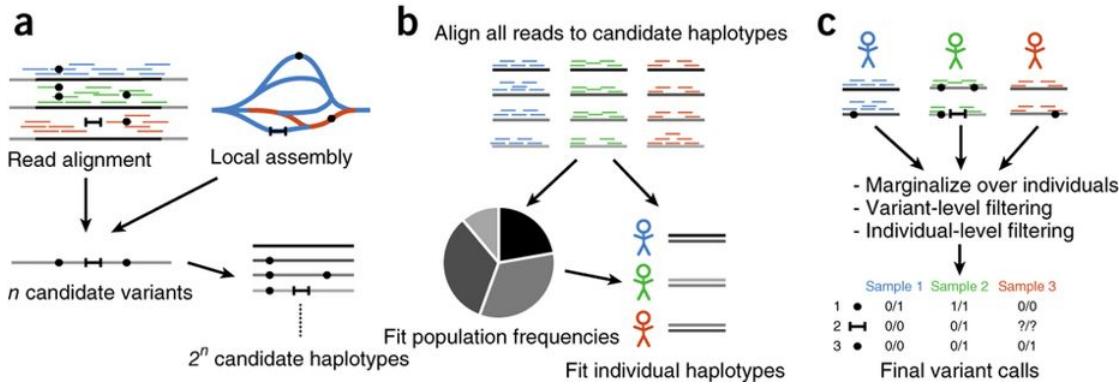


Figure 2.27: Three stages of data processing in Platypus[180]

Platypus works by loading batches of reads into memory from a BAM file. 100 kb (kilobases) are loaded at a time. Low quality reads are filtered out. Sites that are flagged as potential variants by the aligned become candidate variants. The local assembly step operates on successive windows of 1.5 kb in size, including all reads that map to the window as well as their mates, irrespective of where or if they map. As a first step, the algorithm builds a colored de Bruijn graph[101] from the reads and the reference sequence. Candidate variant haplotypes are produced by finding paths that diverge from the reference and come back to the reference by performing a Depth First Search from the graph node that diverges and until the reference is reached again. All such paths are recorded and contain putative SNPs, MNVs, indels, and larger rearrangements that are up to the window-size in length. A prior of 0.33×10^{-3} is used for SNPs under the assumption that SNPs occur in humans at a density of 10^{-3} per base and every non-reference allele is equally likely. Smaller windows are created by combining groups of candidate variants that are within 15 basepairs of each other, as long as window size is smaller than read-length and there are fewer than 8 variants in a group.

Candidate haplotypes are generated by taking all possible combinations of candidate variants (assuming diploid sample), resulting in 2^n haplotypes. For a window with 8 variants there are 256 possible haplotypes. The haplotype likelihood $P(r|h)$ is calculated by aligning reads to each haplotype using a Hidden Markov Model. Platypus uses the Viterbi algorithm[72] to compute the most likely path through the HMM as an approximation of the actual likelihood as a matter of optimization. After haplotype likelihoods are computed for all combinations of reads and haplotypes Platypus uses Expectation Maximization to estimate the frequency of each haplotype under the following model:

$$\mathcal{L}(R|\{h_i, f_i\}_{i=1\dots a}) = \prod_{samples, s} \sum_{haplotypes, h} f_i f_j \prod_{reads, r \in R_s} \left(\frac{1}{2} p(r|h_i) + \frac{1}{2} p(r|h_j) \right) \quad (2.21)$$

where f_i is frequency of haplotype h_i , a is the number of considered alleles, R is all reads, R_s reads in sample s .

The posterior probability of a variant v given the data is computed as:

$$P(v|R) = \frac{P(v)\mathcal{L}(R|\{h_i, f_i\}_{i=1\dots a})}{P(v)\mathcal{L}(R|\{h_i, f_i\}_{i=1\dots a}) + (1 - P(v))\mathcal{L}(R|\{h_i, \frac{f_i}{1-F_v}\}_{i \in I_v})} \quad (2.22)$$

here the likelihood of data R given all haplotypes is compared to the likelihood of R given those haplotypes that do not include v . I_v is the set of haplotype indices such that h_i does not contain v , and $F_v = \sum_{i \in I_v} f_i$. A variant is called when its posterior exceeds a pre-defined threshold. Genotype likelihoods for a variant are calculated by marginalizing over the genotypes at other variants within the window, and the best likelihood is selected as the genotype.

Remarks

Although samtools, freebayes, GATK, and Platypus are not the only tools that have been successfully used for germline SNP calling in the past 10 years, they have enjoyed sustained popularity and continued use in large scale genomics projects such as 1000 Genomes Project, ExAC[119], PCAWG[204] and others. They thus can serve as primary examples of how the problem of calling germline SNPs is currently approached within the field, as well as providing a benchmark for new methods that attempt to solve similar problems. There are several key distinctions in the approaches that these tools take to the overall problem, yet a number of key similarities exist that we outline here and adopt as a general framework for tackling the SNP calling problem in our work.

Site Selection The problem of selecting sites in the genome for detecting variants has variety of approaches from evaluating every single site independently as one that potentially harbors a variant (as in samtools) to the varied windowing approaches used by the other variant callers. Looking at sites independently has the benefit of simplicity, while the windowing approach, at the expense of additional computation, allows a more comprehensive evaluation of a genomic region that is not limited to SNPs but can also support the detection of other types of variants. Thus, even though an independent site approach may be acceptable in an initial implementation, some form of windowing is desired in order to benefit from knowledge about the surrounding sequence context. In general, the interestingness of a site for the purposes of detecting a potential variant is universally linked to the presence of high quality reads spanning a particular locus that disagree with the reference sequence at that locus.

Haplotype Construction Samtools has the simplest model here as it does not attempt to construct haplotypes at all. This limits the ability of samtools to accurately represent genomic architecture, and prevents it from being able to supply

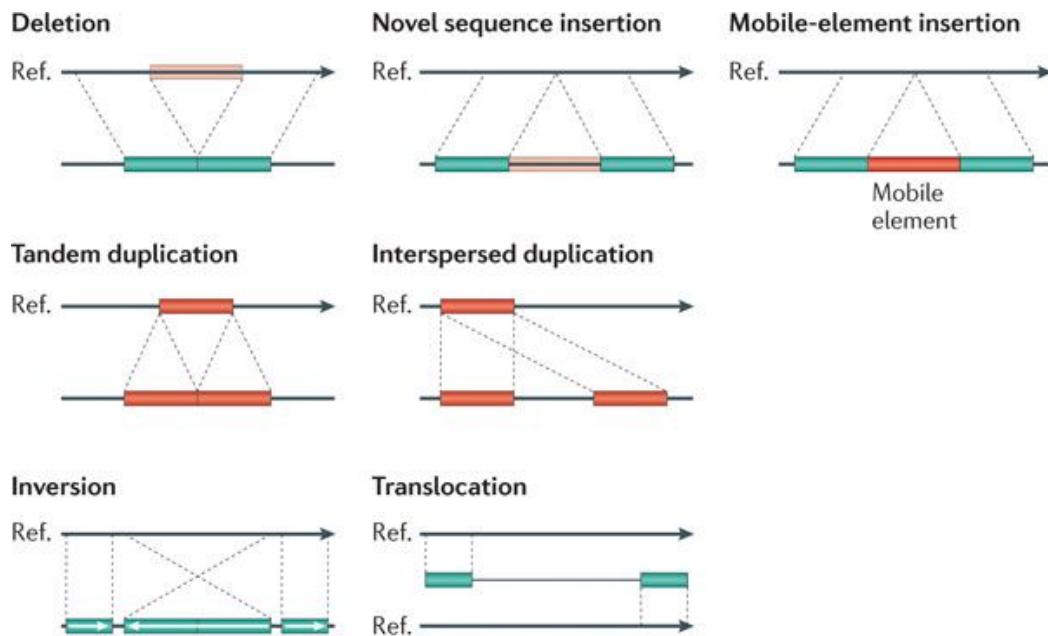
phasing information for variants, which is of interest. Freebayes has the next simplest model, where putative haplotypes are constructed directly from the observed read sequences with the observation window. GATK and Platypus actually perform local assembly in the window to come up with an arrangement of reads that is free of artifacts[129] associated with alignment to the reference. Although local assembly appears to improve the ability of these callers to accurately represent sequence variants (especially non-SNPs) this processing step introduces a significant impact on the overall processing cost, mostly incurred from the resource-intensive HMM-aided alignment of reads to the putative haplotypes.

Allele Frequency Spectrum Estimation A distribution of allele frequencies in a population is of interest as it can be used as a prior in the calculation of genotype likelihoods for the samples under analysis and local and population-specific allele frequencies can vary significantly from values implied by generic population-genetics models. Most callers provide capabilities for estimating the Allele Frequency Spectrum by Expectation Maximization or similar approaches (Equation (5) in [121], not covered here, equation 2.18 for freebayes above). Alternatively, a non-informative prior (as in GATK) can be used.

Genotyping Pretty universally across the methods, genotyping is set up as a Bayesian inference selecting a Maximum Likelihood Estimate or a Maximum A Posteriori estimate of the genotype (for a single site or a haplotype) given the reads data, and taking into account the probability of errors derived from read base quality and mapping quality scores supplied by the aligner. See Eq. 2.10 for samtools', Eq. 2.13 freebayes', Eq. 2.19 GATK's, Eq. 2.21 Platypus' model setups. This model thus remains highly relevant for any new development in the space.

2.2.5 Germline Structural Variant Calling

Structural Variants (SVs) are medium- to large-size alterations (typically >50bp in length) of the genomic sequence that fall into a several broad categories (see Figure 2.28) including insertions, deletions, tandem and interspersed duplications, inversions, translocations, mobile element insertions, as well as complex rearrangements that constitute a combination of the above classes or are otherwise difficult to classify.



Nature Reviews | Genetics

Figure 2.28: Classes of structural variation.[14].

Accurate detection of SVs remains an open challenge since both sensitivity and specificity performance of SV calling is an order of magnitude worse than for SNP calling. For instance, in the latest data release of the 1000 Genomes Project an estimated sensitivity of 88% was achieved for deletions, 65% for duplications, 32% for inversions[208] and False Discovery Rate of up to 89% autocitemills2011mapping. A key challenge that impacts calling accuracy is small read size relative to the size of the variants. Since SVs can be hundreds of kilobases in length and typical Illumina reads are only 150-500 basepairs long, accurate reconstruction relies on an agglomeration of reads in order to produce the variants. Structural Variant detection approaches typically make use of several sources of evidence (see Figure 2.29) for the detection of "breakpoints" - regions of the genome where a DNA double-strand break is detected and sequence is either inserted or excised. The breakpoints are then interpreted to produce the most likely variants that they represent.

Discordantly-mapped read pairs are pairs of sequencing reads that the alignment algorithm has mapped at a distance that is statistically significantly larger or smaller than the average read-pair distance for that sample, or that have an unusual read orientation, since read-pairs are supposed to be sequencing from two ends of a molecule towards each other. Reads that map closer than they are supposed to are indicative of an insertion, reads that map farther are indicative of a deletion, and reads that map in an unusual orientation are indicative of an inversion.

Split-reads are read-pairs where one of the reads maps properly but the aligner is not able to map the other read because different pieces of the read map to different locations in the genome indicating that this read spans a breakpoint. Distance

between the split read pieces and their mapping orientation are informative of the type of breakpoint that the read spans.

Read-depth Regions have a higher than normal or lower than normal read depth (count of reads spanning a region) are indicative of increased or reduced copy number respectively. Care must be taken to distinguish actual SVs from areas of repetitive sequence where the same simple sequence pattern is repeated many times. Typically aligners are not able to accurately resolve such areas and map many reads to the same set of coordinates making it appear like a duplication.

Assembly-based approaches perform local assembly in a reference-free manner to reconstruct the variants encoded in the sample sequence that the aligner may struggle with resolving because of a large number or increased complexity of rearrangements.

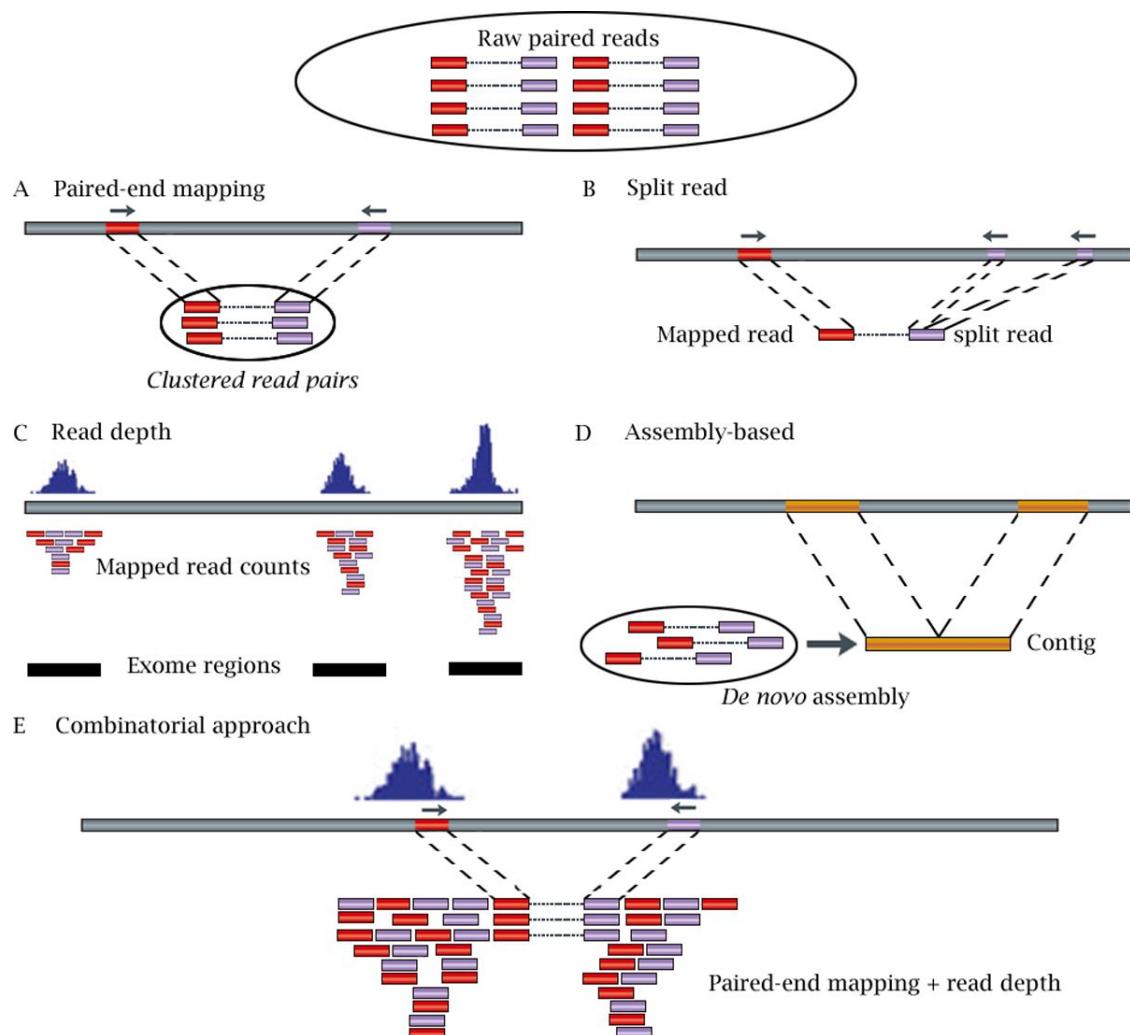


Figure 2.29: Sources of evidence for the presence of Structural Variants.[242].

We look at several SV callers in closer detail.

Delly

Delly[177] is a structural variant calling tool built in 2011 by T. Rausch et al. in the context of the 1000 Genomes Project. Delly uses discordantly-mapped reads and split-reads as sources of evidence for the presence of SVs, is able to characterize a broad set of variants and has been implemented in C++. All of the mathematical formulas and figures in this section are reproduced from the 2012 manuscript by Raush et al.

The discordantly-mapped reads component of Delly begins by computing the median and standard deviation of the read insert size (distance between the ends of the two reads in a pair), as well as the default read orientation by sampling a pre-defined number of reads from the BAM file. Read-pairs that have an insert size farther than 3 standard deviations from the median are considered as evidence for structural variation. This induces a minimum SV size detectable by Delly. See Figure 2.30 for the variant types implied by each insert size deviation and read-pair orientation.

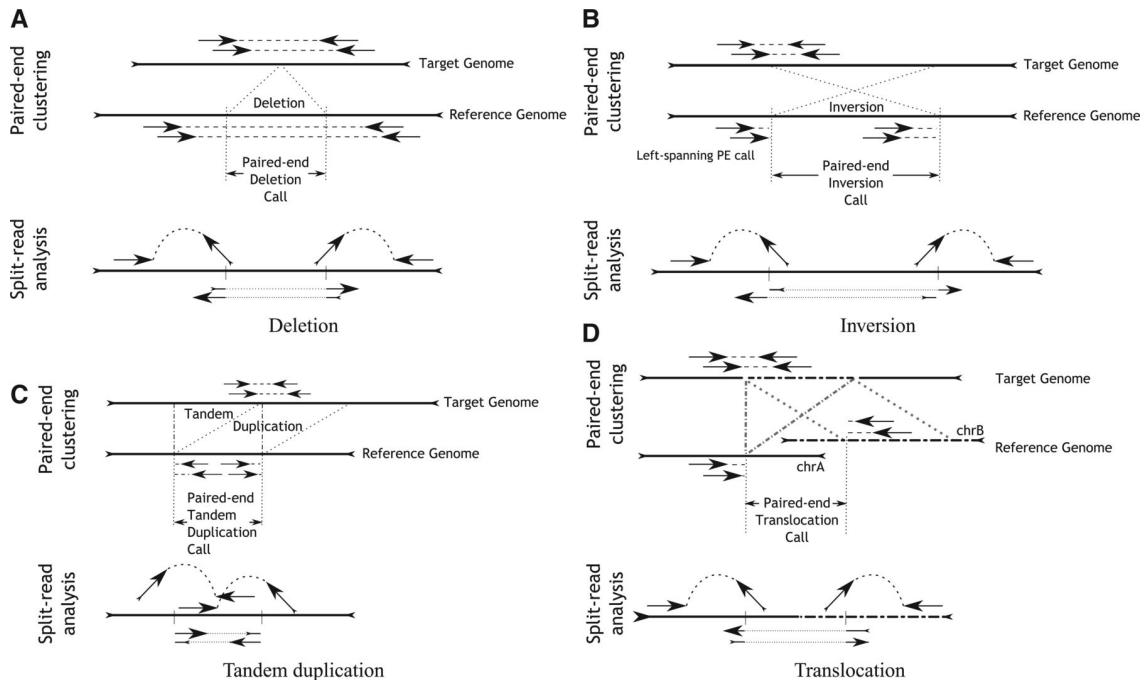


Figure 2.30: Variant classes detectable by Delly based on their read-pair and split-read signatures.[177].

Using the list of discordantly-mapped read-pairs Delly builds an undirected, weighted graph to indicate which read-pairs support the same variant. In the graph $G(V, E)$, a read-pair p_i corresponds to a node $v_i \in V$ and an edge $e_{v_i, v_j} \in E$ connects two nodes that support the same SV. The weight $w(e_{v_i, v_j})$ is the absolute value of the difference between the SV sizes predicted by v_i and v_j . v_i and v_j support the same variant when the corresponding read pairs have the same read orientation and the absolute difference between the left and right ends of the two reads is less than the expected insert size. Variants are identified by computing connected components C_i

of G . When components are not fully connected, Delly sorts the edges in each such component by weight and attempts to find a maximal clique M_i within C_i using edge with the smallest weight as the seed of the clique. The clique is extended by searching for the next smallest edge for which one of the vertices is already in the clique M_i , and requiring that $M_i \cup \{v_l, v_m\}$ is also a clique, until no further vertices can be added. The vertices in M_i are reported as the read pairs supporting that SV. Each rearrangement type is analyzed independently in this manner.

All of the rearrangements identified in the discordantly-mapped read analysis are refined using split-read analysis. The reads used for split-read analysis are those where one read in the pair maps to the reference and the other is unmapped. All such reads within a distance of 2 standard deviations of the median insert size from the breakpoint are considered up to a configurable maximum of 1000. Delly splits each unmapped read into kmers and aligns the kmers to the reference sequence spanning the SV.

Lumpy

Lumpy[117] is an SV caller developed in 2013 by R. Layer et al. and is a package implemented in C++. All of the mathematical formulas and figures presented in this section are reproduced from the 2014 publication by R. Layer et al.

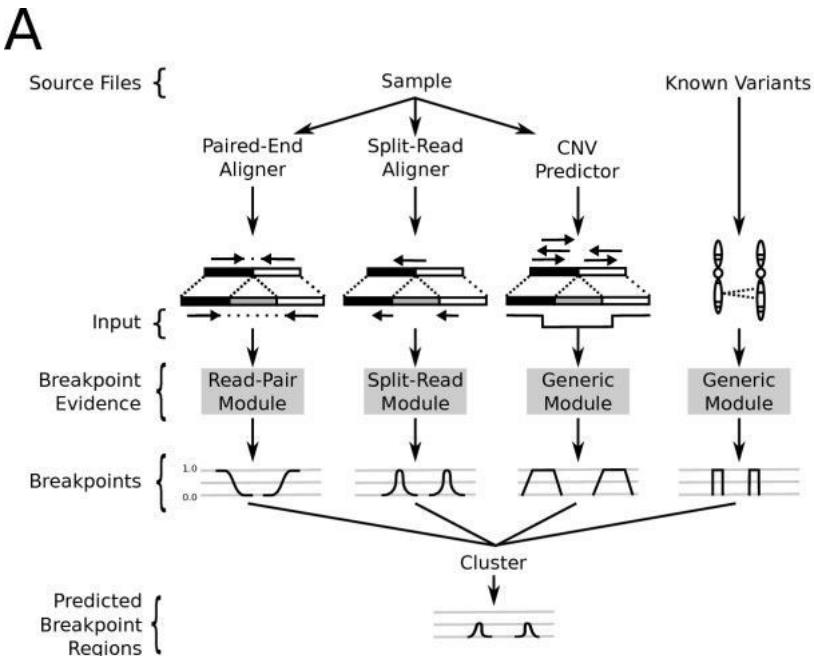


Figure 2.31: Lumpy calling model integrates several signals from a single sample.[117].

Lumpy defines an SV breakpoint as a pair of bases that are adjacent in the sample under study but not in the reference genome. Furthermore, each breakpoint is represented as a pair of probability distributions that span the predicted breakpoint regions and represent the uncertainty about the precise location of the breakpoint.

Lumpy integrates multiple signals (see Figure 2.31) to update the probability distributions that represent each breakpoint based on different kinds of evidence provided. A breakpoint is a tuple $b = \langle E, l, r, v \rangle$ where E is the evidence, $b.l$ and $b.r$ are the left and right intervals each having start and end coordinates, $b.l.s$ and $b.l.e$ for example. $b.l.p$ is a vector of length $b.l.e - b.l.s$ where each $p[i]$ is the probability that $b.l.s + i$ is the true location of the breakpoint. $b.v$ is the breakpoint class (insertion, deletion, etc.). Overlapping breakpoints are merged together. The intervals that contain 95% of the probability density, as well as the Maximum Likelihood Estimates of the location of each variant are returned.

The paired-end analysis looks at read pairs $\langle x, y \rangle$ where each read is aligned to the reference genome as $R(x) = \langle c, o, s, e \rangle$ where c is the chromosome, $o \in +, -$ indicates alignment orientation, s and e represent the alignment start and end respectively. It is assumed that x and y align uniquely and that $R(x).s < R(x).e < R(y).s < R(y).e$. $S(x) = \langle o, s, e \rangle$ is defined to be the alignment of x with respect to the sample's (unknown) genome. Read pairs are assumed to be aligned with $R(x).o = +, R(y).o = -$ and having $R(y).e - R(x).s$ approximately equal to the library preparation fragment size. Read pairs that align outside the expected parameters for orientation and distance constitute evidence for structural variant breakpoints, in particular reads with the same or switched orientation, and pairs that align at a distance shorter or longer than the fragment size. Expected fragment length is estimated from the mean fragment length in the sample, along with its standard deviation. Breakpoint variety is determined as follows:

When $R(x).c = R(y).c$, the variety is inferred from read orientation. $R(x).o = R(y).o$ implies an inversion. $R(x).o = -, R(y).o = +$ implies a tandem duplication, $R(x).o = +, R(y).o = -$ implies a deletion. Insertions are not presently supported. When $R(x).c \neq R(y).c$ the breakpoint is labelled an interchromosomal rearrangement.

$\langle x, y \rangle$ are mapped to $b.l$ and $b.r$ as follows. By convention x is assumed to map to l and y to r . It is assumed that there exists a single breakpoint b between x and y . Orientation of x determines whether l is upstream or downstream from x . Thus, if $R(x).s = +$, then the breakpoint begins after $R(x).e$. The length of the breakpoint interval is proportional to expected fragment length and its standard deviation. The distance of one end of the breakpoint from x is assumed to be less than expected insert size L plus v_{fs} - a configurable number of standard deviations. The probability that position i in the breakpoint interval l is part of the actual breakpoint is estimated as the probability that x and y span i , which is true when the fragment that had produced $\langle x, y \rangle$ is longer than the distance from x to i . Thus, the quantity of interest is $P(S(y).e - S(x).s > i - R(x).s)$ for $R(x).o = +$ and $P(S(y).e - S(x).s > R(x).e - i)$ for $R(x).o = -$. This quantity is estimated from the empirical distribution of fragment sizes collected from the sample.

Lumpy does not perform its own split-read alignment instead relying on the results of a split-read aligner like YAHA[68]. Every split-read is a DNA fragment X that consists of two or more sub-fragments x_i, \dots, x_j that do not map to adjacent locations of the reference. Each sub-fragment pair $\langle x_i, x_{i+1} \rangle$ is evaluated independently depending on how it maps with respect to the reference $R(x)$. When

$R(x_i).o = R(x_{i+1}).o$ the event is marked as either a deletion or a tandem duplication, where $R(x_i).s < R(x_{i+1}).s$ implies a deletion, and otherwise a tandem duplication. When orientations do not match, the event is marked an inversion. To account for potential inaccuracies in the location of the breakpoint with respect to the split-read pair, each split-read pair is mapped to two breakpoint intervals l and r that are centered at the split. The probability is modeled to be highest at the split with an exponential decrease towards the edges, with a configurable width parameter.

SvABA

SvABA[222] is a germline and somatic SV caller by J. Walla et al., created in 2016. Svaba is implemented in C++ and relies on genome-wide local assembly for variant detection. Internally SvABA makes use of SGA[200] and BWA-MEM[131] for assembly and read mapping respectively.

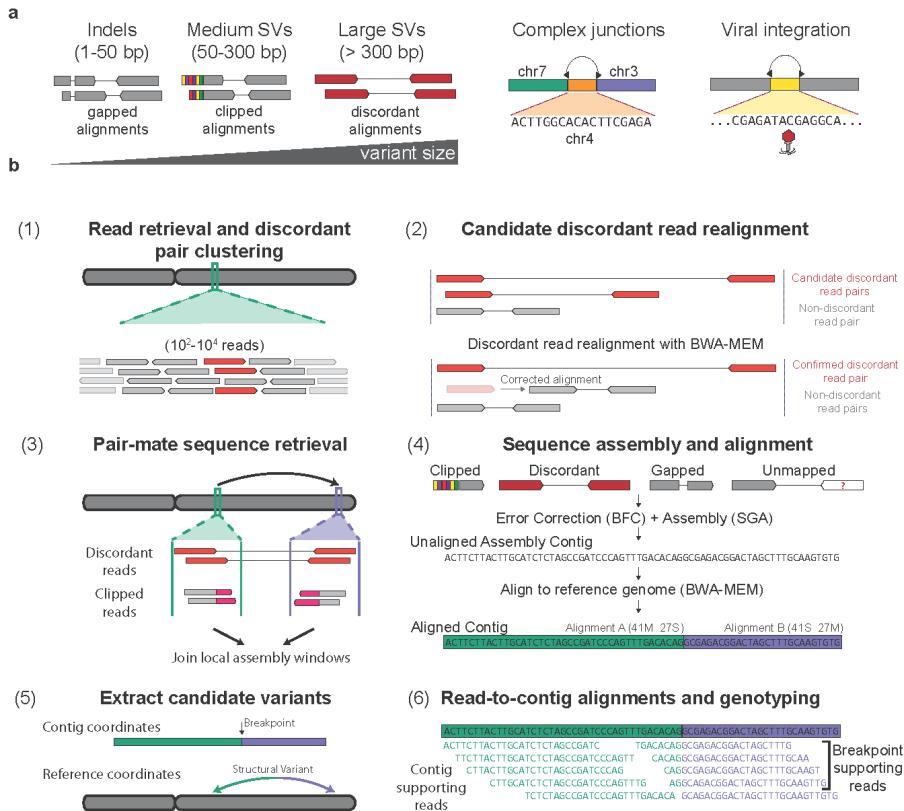


Figure 2.32: The SvABA variant calling method[222].

SvABA performs local de-novo assembly in 25-kbp windows tiled across the genome with a 2kb overlap. First reads that may be indicative of variation are extracted from a BAM file, these include reads with high-quality soft-clipped bases, discordant reads (with non-standard orientation or with insert size greater than four standard deviations away from the sample mean insert size, determined using a sample of 5 million reads with top and bottom 5% of insert sizes truncated), unmapped reads, reads with unmapped mates, and reads with insertions or deletions

indicated in the CIGAR string. Low quality reads that are marked as PCR duplicates, have failed QC, and reads with repeats of >20 basepairs are filtered out. Discordant reads are realigned to the reference with BWA-MEM and those with an available nondiscordant alignment of >70% of the maximum alignment score are discarded. Reads with many different (>20) high quality candidate alignments are also discarded. Remaining discordant reads are clustered based on orientation and insert-size and assembled using SGA.

The contigs produced by SGA are aligned to the reference genome using BWA-MEM and examined for potential variants. Contigs with an alignment that has fewer than 30 nonaligned bases and no alignment gaps are considered reference. Indels are called when the alignment has gaps, and SVs called when the resulting alignment is multi-part. In order to evaluate read support for the putative variants, reads from the windows are aligned to both the reference and the assembled contigs using BWA-MEM. Reads are considered matching to the contig when the alignment score for the read to the contig is greater than the alignment of the read to the reference and is >90% of the length of the match. Reads that have an alignment that is up to 8 bases to the left or right of a putative variant are considered supporting the variant.

Remarks

We looked at Delly, Lumpy, and SvABA in the context of germline SV calling. Although there are some differences in the details of the approaches taken by the tools' authors there are consistent similarities as well, which additionally carry over to other SV callers such as Pindel[237], and Manta[30], that have not been presented here. These approaches rely heavily on paired-end reads and select those that are mapped uncommonly far apart or close together. These are clustered or optionally locally assembled to produce putative breakpoints. Breakpoint locations are further refined by split-read analysis, using those reads that are unmapped by regular read mapping software. These reads are broken down into kmers and each kmer is aligned separately to non-adjacent locations in the genome. Alternative haplotypes may be constructed using these reads and alignments to the reference and the alternative haplotypes scored for genotyping purposes. A new method for structural variant calling should thus focus on making the best use of these two data types (discordant and split reads), while possibly also making use of read depth information to be competitive with the current generation of best callers.

2.3 High Performance , High Throughput, and Cloud Computing

The practice of performing large scale scientific computation on supercomputers or clusters of commodity hardware can be split into two notions - High Performance Computing (HPC) and High Throughput Computing (HTC).

The European Grid Infrastructure defines these as follows[79]:

HPC - A computing paradigm that focuses on the efficient execution of compute intensive, tightly-coupled tasks. Given the high parallel communication requirements, the tasks are typically executed on low latency interconnects which makes it possible to share data very rapidly between a large numbers of processors working on the same problem. HPC systems are delivered through low latency clusters and supercomputers and are typically optimised to maximise the number of operations per seconds. The typical metrics are FLOPS, tasks/s, I/O rates.

HTC - A computing paradigm that focuses on the efficient execution of a large number of loosely-coupled tasks. Given the minimal parallel communication requirements, the tasks can be executed on clusters or physically distributed resources using grid technologies. HTC systems are typically optimised to maximise the throughput over a long period of time and a typical metric is jobs per month or year.

Although early High Performance Computing efforts (1960's - 1980's) relied on supercomputers with a shared memory model[186], where all of the memory was shared between multiple processors, by the late 1980's machines with a distributed memory model[161], where each processor has its own memory, started gaining ground, forming the basis for the modern HPC cluster.

The software interface that the user has to a HPC/HTC cluster typically takes the shape of a queueing system such as PBS[91] or LSF[244] where the user writes a script that submits a series of jobs to the queueing system. The jobs can invoke software that is installed by the IT department that manages the cluster. The user is not able to install any software and has limited visibility into the runtime performance characteristics of the jobs they submit.

Cloud computing has emerged in the early 2000's enabled by improvements in hardware virtualization which was driven by the adoption of Virtual Private Networks, and the desire to commercialize access to compute capacity as a utility[28].

The National Institute of Standards and Technology provides a standard definition of cloud computing that encompasses several areas of this domain - Essential Characteristics, Service Models, and Deployment Models[151].

The Essential Characteristics of a cloud are as follows:

On-demand self-service - End-user can independently manage infrastructure without involving the service provider.

Broad network access - Cloud resources are available on the network via a set of standard protocols.

Resource pooling - Service providers dynamically assign virtual infrastructure in a multi-tenant environment based on consumer demand.

Rapid elasticity - Resources can be elastically provisioned and discarded according to customer requirements.

Measured service - Resource usage by end users is measured and transparently provided back to the user by the service provider.

It is the self-service and broad network access characteristics that set cloud computing apart from traditional HPC computing the most.

Service Models include:

Infrastructure as a Service (IaaS) - This service allows the user to provision and control virtualized infrastructure such as VMs and networks.

Platform as a Service (PaaS) - This service allows the user to deploy their application onto virtualized hardware but not to control the management of the infrastructure.

Software as a Service (SaaS) - This service allows the user to make use of applications that are deployed on virtualized hardware but not to manage the applications or the infrastructure itself.

The Deployment Models covered by the NIST definition are as follows:

Private Cloud - Operated privately by a single organization and not accessible on a public network.

Community Cloud - Established for use by a particular community of users with a common interest.

Public Cloud - Established for general use by the public.

Hybrid Cloud - A collection of cloud entities that use one of the other deployment models but allow application portability.

The first publicly available commercial cloud computing platform has been developed by Amazon.com and launched in August, 2006 in the form of two services - Elastic Compute Cloud (EC2), and Simple Storage Service (S3). Cloud offerings by Microsoft and Google followed in 2010, and 2012. This early lead has allowed Amazon to capture the majority of the public cloud computing market, earning \$2.57 billion USD in Q1 2016 revenue.

One of the main drawbacks, however, of using Amazon's or another proprietary cloud solution is the issue of "vendor lock-in" i.e. inability to easily switch infrastructure providers should the customer wish to do so, because of the amount of software relying on the proprietary cloud provider protocols. Another key reason for avoiding public clouds is the necessity to store sensitive data. This issue applies both to the commercial enterprise (with industries such as banking, and payments) and scientific domains (especially genomics and medicine) where handling of sensitive patient data is restricted based on both technical security, as well as ethical considerations[108].

To help alleviate these concerns an open-source cloud platform called Openstack was launched in 2010 jointly by Rackspace Hosting and NASA[193]. Openstack provides most of the same features that are provided by Amazon Web Services and other commercial cloud providers as free open-source tools. These include:

- Infrastructure
- Networking
- Identity Management
- Block Storage
- Object Storage
- Managed Databases
- Queues
- Monitoring

Openstack deployments form the basis for most academic private and community clouds such as EBI Embassy Cloud[42], University of Chicago Open Science Data Cloud[85], Cancer Genome Collaboratory[238], and Helix Nebula[147]. Because these clouds implement the security measures necessary when handling patient data they are a system of choice for large scale bioinformatics analyses.

2.4 Workflow Systems

The focus on workflow stems from the work of Frederick Taylor (1856-1915) and Henry Gantt (1861-1919) on the improvement and automation of industrial processes, also known as "scientific management"[210]. One of the key techniques that were devised at the time and served as the prototype for future workflows were "time and motion studies"[20] where employees were observed as they performed repetitive cycles of work in order to determine standard execution times and sequences of steps. As this field evolved over the course of the 20th century it gave rise to several other related fields such as Operations Management, Business Process Management, and Lean Manufacturing.

In 1993 an international consortium was formed with the purpose of defining the standards related to workflows and workflow management systems. This consortium is called the Workflow Management Coalition (WfMC). One of the key specifications produced by the WfMC in 1995 is The Workflow Reference Model[94]. This document provides two basic definitions that illuminate the scope and purpose of workflow systems:

Workflow - The computerised facilitation or automation of a business process, in whole or part.

Workflow Management System - A system that completely defines, manages and executes "workflows" through the execution of software whose order of execution is driven by a computer representation of the workflow logic.

A number of standards have been produced for workflow definition, many of them are XML-based[195]. Notable examples include:

XPDL - Was developed by the WfMC, currently at version 2.2, as of 2012. Uses an XML dialect to express process definitions

BPML - Developed by the Object Management Group (OMG) using XML. Deprecated as of 2008 in favour of BPEL.

BPEL/BPEL4WS - Developed by Organization for the Advancement of Structure Information Standard (OASIS). Uses XML format. Adopted by Microsoft and IBM for their workflow products -

Graphically, workflow definitions are typically expressed using a Petri-Net[172] or Business Process Model and Notation (BPMN), the latter borrowing its structure from UML activity diagrams. A set of workflow definition design patterns exists to guide workflow creation[55]. A workflow engine is responsible for ingesting workflow definitions, generating their graphical representation, and allowing the user to execute the workflow definitions on suitable hardware.

As initially the focus of workflow systems research and development has been on process improvement within commercial enterprises there exists a large pool of workflow engine implementations targeted at that sector. Some of these are:

jBPM - An open-source workflow engine that is based on the Java platform and is currently owned by Red Hat.

Activiti - An open-source workflow engine that has been developed by previous jBPM developers.

Oracle BPEL Process Manager - A commercial workflow engine acquired by Oracle from Collaxa in 2004, now integrated into the rest of the Oracle portfolio.

Websphere Process Server - Commercial workflow engine that is part of IBM's Business Process Manager suite.

Although these tools have gained wide adoption in the enterprise community they have had limited success within scientific circles. Instead, several open-source workflow management systems exist that have been purpose-built for the scientific domain, and especially bioinformatics. These include:

Kepler[138] - A Java-based WfMS built on top of the Ptolemy II[52] execution engine.

Taverna[164] - A Java-based WfMS originally built by myGrid, currently under incubation at Apache Software Foundation.

Galaxy[80] - A Python-based WfMS developed specifically for bioinformatics applications with a focus on GUI-driven development of workflows.

Curcin et al[46] provide a head-to-head comparison of six scientific workflow systems including Taverna and Kepler, whereby Taverna is described as primarily being aimed at researchers who wish to build scientific workflows from web services utilizing a proprietary XML dialect called SCUFL which implements a DAG model of workflows. The primary execution environment for a Taverna workflow is on a grid or an HPC cluster. Kepler implemented a different methodology, whereby workflow modelling, which is taken on by Actors, is separated from workflow execution, taken on by Directors. An Actor knows only about its inputs, the computation that it needs to perform, and the output that it needs to produce, while Directors provide different models of execution, such as Synchronous Data Flow, Process Network, Continuous Time, and Discrete Event.

The Galaxy workflow framework has a specific focus on bioinformatics analyses and comes with a large library of community-developed bioinformatics workflows. The user creates and executes workflows via a web-based GUI where pre-installed tools and scripts can be laid out into a pipeline. The primary deployment environment for Galaxy is on an institutional HPC cluster although a separate component allows the deployment of a Galaxy instance on Amazon Web Services[10].

Chapter 3

The Butler Framework - Requirements and Architecture

In this chapter we specify a set of requirements for a large-scale cloud-based scientific workflow framework. The chapter is split into two sections - Section 3.1 Functional Requirements deals with requirements for how to access data, install software, manage workflows, and troubleshoot errors, Section 3.2 Non-functional Requirements deals with issues of Scalability, Availability, Ease-of-use, and Interoperability.

3.1 Functional Requirements

Running scientific analyses requires the following broad set of capabilities:

- Access to data
- Access to compute capacity
- Implementations of one or more scientific algorithms
- A workflow that defines the sequence of steps in the analysis
- A workflow engine that handles job scheduling and execution
- A system of record for what analyses have been performed
- A set of tools for troubleshooting error conditions

Operating such a system on the cloud necessitates an extra set of capabilities that enable users to take advantage of the scalability and elasticity offered by cloud computing, while retaining cost effectiveness and security. These capabilities include:

- Provisioning of cloud infrastructure

- Configuration of virtualized hardware
- Service discovery

3.1.1 Access to Data

Scientific analysis typically requires access to data files to run various tools on, thus an analysis system needs to provide a mechanism for accessing data. Depending on the architecture of the system in question, several data sources can be identified, each of which stipulates a particular data access mechanism. These include:

- Data stored in a third party data repository on the internet
- Data stored on a network accessible shared storage folder
- Data stored on cloud specific Block Volumes (such as Amazon's EBS, or Openstack Cinder)
- Data stored on cloud specific Object Storage Services[66] (such as Amazon S3, or Google Cloud Storage)

3rd Party Repository Data

Bioinformatics, like many fields of science, has a vast number of data repositories and reference data sets available over the Internet, in both, free access, and authenticated modes. The method of access to these services is typically specific to each repository, although is frequently limited to HTTP and FTP protocols. Thus, a cloud-based system that allows access to external IP ranges via HTTP and FTP should be able to meet this requirement to a sufficient degree.

Network Accessible Shared Storage

A large data repository that is hosted within the same data center as the compute cluster is the data access method of choice within HPC deployments, but is also used within cloud computing environments, especially private academic clouds. A distributed network accessible file-system such as Isilon OneFS, Lustre, GlusterFS, MooseFS, GFS, or HDFS is typically used[191]. In order to take advantage of such a shared file-system a cloud-based compute cluster simply needs to have mount privileges on the cluster virtual machines. Once mounted, the file-system can be accessed as if it was a local file-system. It is typically the case that full root is available on cloud-based VMs, so this method of access remains both popular and well supported, especially for analyses where multiple VMs may need to access the same file at the same time.

This approach has three key drawbacks:

- Shared access to a file server can run into scalability bottlenecks at the storage and network layers as the number of VMs simultaneously accessing the resources increases. High performance storage providers such as Netapp and Isilon can support simultaneous transfer rates of up to 40 GB/s, which allows a significant number of VMs to simultaneously access the shared resources, however, as shown in the Experimental Validation section, even 1000 compute cores can easily consume over 25% of that bandwidth, thus limiting the overall system scalability to about 4000 cores.
- While academic cloud providers are frequently running a large scale shared storage server in support of their HPC environments, commercial cloud providers do not have this service out of the box, thus it is up to the user/operator to set up such a shared file-system based on VMs and block-storage volumes (and possibly ephemeral disks), which can be a significant cost.
- Implementing data access security is challenging, because once a shared file-system is mounted, data access is granted based on Unix groups without checking credential validity with the data owner.

Block-level storage

Most clouds provide a block storage service (Amazon, Google, Microsoft, Openstack, and others). Block storage is different from typical hard-drives that are available as a standard together with Virtual Machines, in that the standard hard-drives are considered "ephemeral" storage - their contents are available only for the lifetime of the VM that they are attached to. Due to the short lifetime, such storage is typically only acceptable for use as scratch-space, and not for the long-term storage of data under analysis. Block storage volumes offer an alternative, whereby a block storage device can be attached and detached to any VM within the same data center without the loss of information. Once attached, the block storage volume can be mounted as if it was normal local storage.

For the purpose of scientific analysis on the cloud, block-storage offers an attractive option whereby a data set can be prepared and staged on a block storage volume outside the scope of a particular analysis, and can then be mounted on a VM that will perform the analysis, as well as being reusable for other analyses downstream.

Key drawbacks of this approach are:

- The same block volume cannot be mounted by several VMs at once, thus causing data duplication, or ruling out this data access method, where simultaneous access to a file by several VMs is required.
- Limited size of a single block storage volume (currently 16TB on Amazon for instance).
- Higher storage cost than object storage.

Block level storage is automatically available in all cloud environments where such a service is present and it is up to individual analyses to take advantage of this method of data access.

Object Storage

Most cloud products on the market today provide an object storage service. Examples of this are Amazon S3, Openstack Swift, Ceph[226], and Microsoft Blob storage. Object storage provides a highly scalable alternative to shared filesystems and block storage volumes, where each object of interest is stored in a "bucket" and can be retrieved by its identifier. This method of access is especially attractive because data access security can be implemented on an individual object basis, something that is difficult to implement with other data access methods. Thus, scientific analyses that operate on sensitive data, such as those performed for the biomedical field with human subjects can greatly benefit from adopting this method of access for cloud-based analysis. Additional benefits of object storage include virtually limitless scalability of storage space, and low cost, relative to other storage methods.

A drawback of this approach is that object storage based systems do not function in the manner of POSIX compliant file systems that many are familiar with. Thus, an analysis framework that wants to support object storage as a method of data access needs to provide support for managing user credentials, retrieving data of interest by identifier from the object storage and into a scratch space available to the VM (such as ephemeral disk or block storage volume), writing intermediate analysis results back to the scratch disk, and storing the final analysis results back into object storage based on a predetermined bucket structure.

3.1.2 Access to Compute Capacity

Running analyses requires access to computational resources such as CPU and RAM and doing so on the cloud is significantly different from the way the same goal is accomplished in traditional HPC environments.

A traditional HPC environment has a static pool of resources, access to which is facilitated via a queueing system. All users submit their jobs to a priority queue and each job is eventually scheduled to run on some server. The server is a long running machine, which is not dismantled between jobs, and the user has little control over the server's configuration in terms of software or hardware. If the HPC data center operates in a cost-sharing model, costs are apportioned based on either a fixed allocation between departments or research groups, or based on resources used during the job execution.

Because access to cloud-based resources is charged based on time-in-use i.e. by the hour, or by the minute, running analyses on the cloud requires a different mode of operation in order to remain cost effective. Virtual Machines need to be created from scratch and used only for the duration of time strictly required by the analysis,

at which point they need to be destroyed. Furthermore, because the cost of each Virtual Machine is directly related to the amount and type of resources that it consumes, and the user has complete control over this configuration, it is in the user's interest to optimize the hardware configuration of each VM such that it fits the type of analysis being performed as some analyses will benefit from higher CPU, RAM, optimized disk I/O, etc. A successful framework for cloud-based analysis then needs to provide the following capabilities in the area of Access to Compute Capacity:

- Ability to authenticate and interact with multiple cloud-provider APIs
- Ability to define hardware configurations of Virtual Machines.
- Ability to easily create and destroy Virtual Machines based on pre-defined hardware profiles.
- Ability to specify network topology and security rules.

Interact With Multiple Cloud-Provider APIs

Because there are many cloud providers out there, each with certain advantages and disadvantages when it comes to features and cost it would be beneficial for a generic analysis framework to be able to interact with as many cloud provider APIs as possible. This is challenging because different cloud APIs are in-general incompatible although similar in nature, and it is necessary to provide a translation layer, so that the users of the framework do not have to learn different API dialects depending on which environment they wish to deploy to. Authentication methods typically rely on a known URL, username/password, and a public/private key pair, and thus are fairly amenable to standardization.

Define Hardware Configurations for VMs

Because each type of analysis may require several flavours of VMs, each with a different hardware configuration, it would be beneficial for the user if the analysis framework provided a mechanism to easily define such configurations in a human readable file that can be versioned and source controlled. Key fields that need to be captured by such a configuration file include:

- VM naming template
- Number of VMs
- Number of CPUs
- Amount of RAM
- Number and type of hard disks

- Security Groups
- SSH Keys for logon
- Initial machine image or snapshot to use (to avoid starting from a completely empty VM)
- Network configuration (IP address, subnet, floating IP)
- Post-initialization commands (such as registering with a master node or updating packages)

Create and Destroy Virtual Machines

Due to the dynamic nature of cloud computing clusters where Virtual Machines are frequently created and destroyed it is necessary to provide a convenient method for carrying out both of these operations on a large number of VM instances and in a cloud agnostic manner. The user should be easily able to dispose of any number of running machine instances by name, as well as being able to create any number of new machines based on the templates described in the previous section

Define Network Topology and Security Rules

While in an HPC system the network topology and security are completely specified by the IT group, it is up to the user to adequately specify both in a cloud computing environment. Thus, a user needs to be able to specify networks, traffic routing rules, and control access to resources within the network based on port, protocol, and address of the requester. This requirement is especially important when it comes to handling and managing human genomic data which is highly confidential - a typical network and security configuration that needs to be supported is one where a central network router controls access to and within the cloud tenant. Only a limited number of known IP addresses from the internet are allowed inbound access over a secured protocol. Furthermore, Virtual Machines within the cluster are completely locked down except for a limited number of ports and protocols, as necessitated by their role within the cluster.

Because cloud computing clusters are frequently created and torn down it is necessary to be able to express the network and security configuration as a set of rules that can be easily re-instantiated as needed.

3.1.3 Implementations of Scientific Algorithms

Running scientific algorithms en-masse is the key requirement for a scientific workflow management system, and even though the implementation of the algorithms themselves is outside of the scope of such a framework, existing algorithm implementations need to be easily brought into the system and configured for running. In an

HPC environment tool installation is performed by the IT department managing the HPC cluster, but in a cloud computing environment it falls upon the user to carry out installation of all software, thus a system of support is required for managing installations of scientific software on Virtual Machines in the cloud.

Several mechanisms for installing software on VMs exist and need to be supported:

- Installation of a binary file from a known URL
- Compilation of a tool from source code
- Installation of a Virtual Machine image
- Installation of a lightweight container image

Installation of a Binary File

Oftentimes tool authors provide compiled versions of their software on the Internet. These are typically available for major operating systems. If such a tool does not have other dependencies, then installing it simply requires access to a particular URL on the Internet from the VM that the tool needs to be installed on. From a security perspective this means allowing outgoing traffic from VMs inside the cluster for protocols such as HTTP, HTTPS, FTP, SFTP, FTPS and their associated ports. Installation then proceeds by retrieving the required binary resource and possibly setting up some symlinks for convenient invocation.

Compiling From Source Code

A large proportion of scientific software exists as source code available over the Internet on sites like Github, Bitbucket, Sourceforge, etc. along with instructions for compiling the software on a supported platform. This provides a major avenue for installation of scientific software and requires capabilities for downloading the source code from the internet via a tool like git, along with possible dependencies, and then executing a build script provided by the tool author, thus requiring the same capabilities as specified for installing binary files, as well as possibly elevated user privileges for installing services or adding users.

Installation of a VM Image

Several widely used formats of VM Image exist, such as OVA, OVF, VMDK, AMI, etc. A tool author may choose to make their algorithm available as a complete VM image that can be instantiated within a cloud computing cluster. This is especially true for complex applications that have many dependencies and very specific OS requirements that are hard to replicate reliably. The process of instantiating VM images is typically similar between cloud providers although differences exist.

Installation of a Lightweight Container

A more lightweight approach than using entire VM images is shipping software via a lightweight container such as Docker[152]. This method of software distribution is gaining in popularity in the last two years. Lightweight container images are typically smaller than entire VM images as they only concern themselves with providing an application runtime environment, rather than an entire VM emulation, and they typically allow the running of multiple such containers simultaneously on a single VM in a micro-services container-based architecture.

In order to support this method of distribution the cloud computing cluster needs to be running a container management platform such as Docker Swarm or Google Kubernetes.

3.1.4 Workflow Definitions

A typical scientific analysis consists of a series of steps where a set of input files (or samples) are transformed through, possibly multiple, computational stages to produce a set of output files that may be used as is, or retained for further analysis.

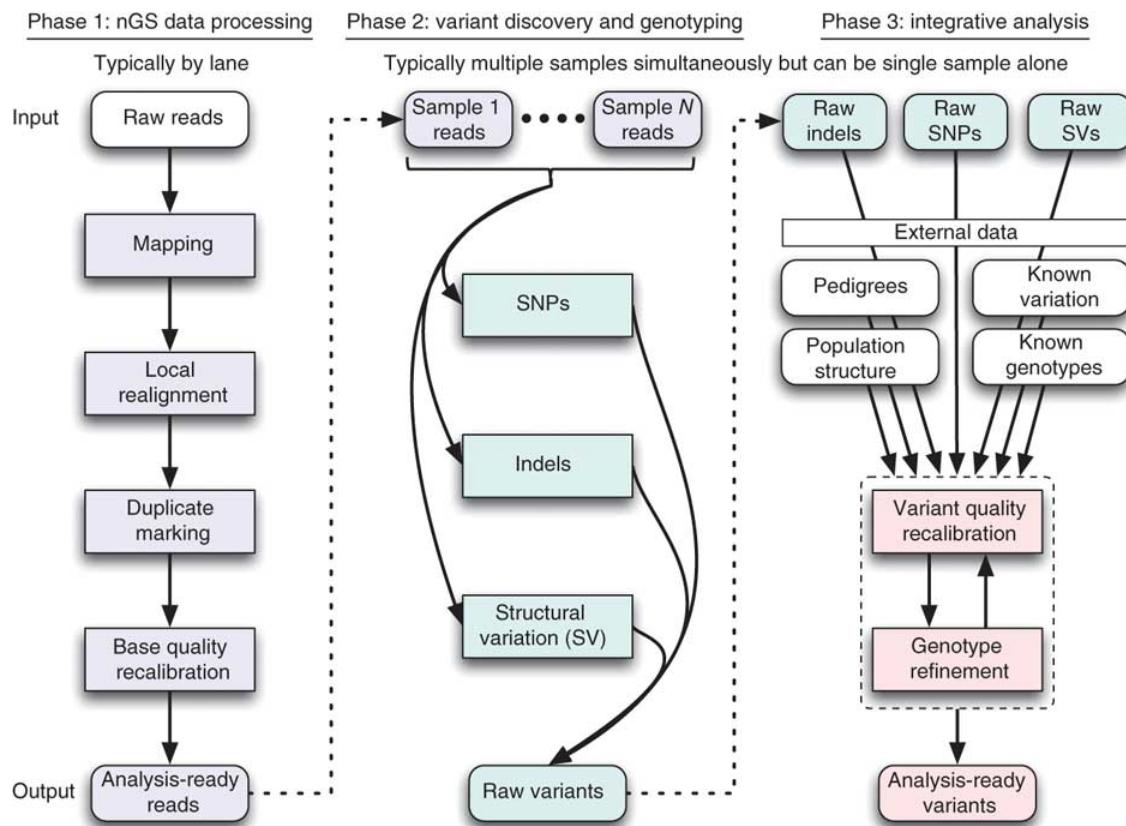


Figure 3.1: NGS Workflow[54].

To be able to reliably carry out such analyses it is desirable for a system to have a number of capabilities that define the "workflow" of execution steps. These are:

- Define a structure that encodes a sequence of steps and the conditions for moving from one step to the next.
- Ensure deterministic behaviour where possible i.e. same inputs produce the same output when run multiple times (whether in the same computational environment or different environments).
- Ensure that the analysis has a finite run-time where possible.
- Allow fine-grained control over the analysis configuration.

Workflow Structure Definition

An effective and commonly utilized method of representing a workflow in the scientific analysis and other contexts is as a directed graph. Each vertex in the graph represents a particular computational step that needs to be carried out, and graph edges represent, possibly conditional, transitions between the vertices. This representational method allows the user to comprehensively express both the steps involved in an analysis, as well as specifying their sequence and executing control flow. Furthermore, the graph-based approach allows for a graphical representation of the workflow structure that is readily comprehensible by humans, thus increasing its utility. In general, the following requirements need to be met in order for the graph-based workflow representation to be fit-for-use in the scientific analysis context:

- The workflow should be encoded using a language and format that is easily readable by humans.
- It should be possible to pass parameters to a workflow – values that can be used at runtime to affect workflow behavior.
- A workflow state should be able to invoke any program that is installed on the machine that is running that workflow state.
- A workflow state should be able to interrogate the environment that it is running on i.e. check for presence/absence of certain files, communicate with a database, access URLs on the Internet.
- Any number of transitions should be able to enter or leave a state.
- It should be possible to render a graphical representation of the workflow as an image.
- It should be possible for different states in a workflow to exchange information.

Deterministic Behaviour

Scientific reproducibility is a key concern that needs to be maintained in order for the system to be usable in the context of scientific analysis. Reproducibility, specifically, refers to the requirement for a user to be able to produce the exact same result as reported by another user using the same workflow definition, input files, and computing environment. While certain algorithms that may be executed as part of a workflow state like Expectation Maximization[155], or Stochastic Gradient Descent[24] behave in a non-deterministic manner where exact reproducibility of results may not be possible, the workflow system itself should not introduce any stochastic components when encoding workflow structure.

Finite Run-Time

The directed graph structure allows for transitions back to a previously visited vertex, thus creating graph cycles. Although this improves the expressiveness of the structure it is generally undesirable in the context of workflows as such a workflow may end up in a perpetual loop within the cycle requiring human intervention to diagnose and rectify at run-time. Although it is generally impossible to guarantee a finite run-time for a workflow due to the possibility of any underlying computational algorithm itself getting stuck in a perpetual loop or resource deadlock, we would like to rule out such a possibility at least from the perspective of workflow structure. Thus, we place an additional constraint on the workflow definition to be a Directed Acyclic Graph (DAG). Although this complicates somewhat the encoding of use cases where certain tasks need to be repeated a number of times, these use cases are generally still attainable via programmatic generation of a series of states or using sub-graphs and gives us the added comfort of ruling out infinite loops where possible.

Analysis Configuration

A scientific analysis typically needs to be configured and parameterized at multiple levels. When considering different types of parameterization required, 3 distinct levels of parameterization can be identified. These are:

Workflow level – configuration that applies to a particular workflow regardless of which analysis it is used on. This may include things like paths to the location of certain programs used by the workflow, or their general invocation strings, or certain reference values.

Analysis level – this includes values that may be different from one analysis to the next but are not different between samples under the same analysis. Examples of such configurations are: common flags to pass to tool invocations, where to store analysis results, where to look up reference data sets.

Execution level – these parameters differ even from one invocation of a given workflow under the same analysis to the next invocation. The most common

instance of such a parameter is the set of names of input files that need to be processed by the workflow representing one sample.

It is natural to view these configuration levels as a three level hierarchy where Analysis configurations expand on and override Workflow level configurations, and Execution level configurations expand on and override Analysis level configurations.

In order to successfully configure the workflow system it is necessary to be able to specify and permanently store such configurations in a format that is both human- and machine-readable. Thus, a user who is conceiving a particular analysis should be able to author a set of configurations that embody the nature and specifics of the analysis being performed. These configurations should then be easily transferrable to another individual or system instance for reproducibility purposes. Once the configuration of the system is fully specified the user should be able to launch their analysis according to this specification.

At run-time, the workflow system should be able reconstruct one *effective* configuration from all levels of the configuration hierarchy and apply it appropriately.

3.1.5 Workflow Engine

A Workflow Engine is necessary for the set of workflow definitions to be executable on a compute cluster. This workflow engine needs to be able to fulfill the following broad set of requirements:

- Workflow parsing and translation
- Workflow state management
- Workflow scheduling
- Workflow execution

Workflow Parsing and Translation

While a workflow itself encodes the structure and sequence of computational steps that need to be performed, the workflow engine needs to be able to parse such a workflow definition and translate it into a set of commands that can be runnable on a real machine. To aid scalability it is desirable for the workflow to be broken down into a set of tasks at this stage that can be run on different physical machines. The workflow engine then needs to be responsible for producing this set of uncoupled commands from a monolithic workflow definition.

Workflow State Management

Workflow state management encompasses concerns of how to keep track of workflow definitions, including their versioning, as well as the status of various workflow instances. The workflow engine, then, needs to keep a registry of workflow definitions where, along with the workflow code and an identifier, the user can record useful metadata related to that workflow, such as - workflow version, author, owner, creation date, whether the workflow is enabled, etc. This registry is the authoritative source of information on what workflow capabilities a particular deployment of the workflow system has, and every new workflow, or update to an existing workflow must be recorded within this registry.

Furthermore, a particular version of a workflow definition may give rise to any number of workflow instances i.e. particular invocations of the workflow on a set of inputs and within the context of an analysis. Each instance may be in one of a number of different states, such as - Stopped, Running, Queued, Completed, or Failed. The workflow engine is responsible for keeping track of all of the workflow instances for a given workflow, and their states, and is responsible for transitioning the workflow instances from one state to the next, based on the preconditions for each such transition. These state transitions are not to be confused with the state transitions that are encoded within the workflow definition, as the workflow definition state transitions are custom to each workflow and encode the logic of the underlying scientific analysis, whereas the workflow instance state transitions and their conditions are standard for all workflows and workflow instances and describe the general workflow lifecycle. As with the metadata recorded for workflow definitions, the workflow engine should also record metadata associated with workflow instance state transitions, the most important of which is the time-stamping of all such transitions.

Workflow Scheduling

Once a workflow definition is parsed and translated into a series of runnable tasks, the workflow engine needs to be able to schedule these tasks for execution as appropriate. A Scheduler component within the engine needs to be able to match up two sources of information - availability of computational resources, and availability of runnable tasks, to produce a schedule i.e. a set of task-to-resource assignments within an appropriate timeframe. In order to accomplish this the Scheduler needs to be able to carry out a number of tasks, as detailed below.

To ascertain availability of computational resources the Scheduler needs to be able to communicate with all machines within the cloud computing cluster that are designated for running tasks. Furthermore, the scheduler needs to be able to interrogate the state of these machines to determine what the current level of load on each machine is, and whether the machine can accept more load in the form of new tasks. As the load level of each machine is of a dynamic nature, and is based on the completion of currently running tasks, the Scheduler needs to be in a constant state of communication with the entire compute cluster, in order to maintain an

up-to-date picture of resource availability.

To establish a list of runnable tasks the Scheduler needs to iterate over all currently running workflows and determine the state of execution within them. As tasks are completed within each workflow instance, other tasks that are downstream from them in the workflow definition may become runnable. The scheduler should then determine for each workflow instance what its set of currently runnable tasks is, based on the structure of each workflow definition and the current state of the workflow instance.

One important concern within the realm of workflow scheduling is the concept of workflow and task priority. It is natural to think that not all workflows, and not all workflow tasks have the same priority i.e. some are more important than others and should, thus, have precedence when it comes to scheduling. It is then important for the Scheduler to be able to incorporate the concept of priority into the task scheduling decisions that it is making in order to meet user requirements.

Armed with a prioritized list of runnable tasks and a list of available resources the Scheduler needs to produce as set of task-to-resource assignments that can be used for workflow execution.

Workflow Execution

The purpose of a workflow engine, at its core, is to execute workflows, thus, a set of execution capabilities is required. The sections above describe the requirements for parsing and translating a workflow into a set of runnable tasks, managing their state, and scheduling task execution according to its priority and resource requirements. The Execution component of a workflow engine needs to handle the actual running of tasks that have been scheduled.

Each machine in the cloud computing cluster that is part of the workflow system needs to be able to accept from the workflow scheduler a task execution assignment that encodes the details of the actual task that needs to be run. The task itself may consist of any number of computational steps. The Execution component is then responsible for transitioning a task from the Scheduled state to a Completed or Failed state, and carrying out the computational steps encoded within the task definition. This typically involves running other programs, collecting their return statuses, and execution timestamps and relaying this information to the Workflow State Management component.

When the Execution component encounters errors during task runtime it should be able to not only collect comprehensive information about the error condition, but should also be able to retry running the task without necessarily failing the overall workflow instance.

3.1.6 System of Record

One of the key requirements for a workflow system to be used in a scientific context is reproducibility. This concept was referenced earlier when describing the desire for a workflow to behave deterministically where possible. Another important factor that affects reproducibility is the method by which the course of a scientific analysis is encoded. We thus need an accurate system of record that keeps track of the following information:

- The various analyses that are undertaken
- Workflows that are executed as part of each analysis
- Samples that are part of the analysis
- Overall system configuration

For each of the items above it is important to capture and permanently store a number of vital fields such as names, and unique identifiers, owner, version, and timestamp. Then, when the results of such an analysis are used in a scientific publication it is enough to provide the unique identifier of this analysis to be able to recover the technical details that will aid in reproducing analysis results by a 3rd party.

3.1.7 Troubleshooting Errors

One of the inevitabilities of large scale computing is the occurrence of error conditions. The probability of an error being encountered in a unit of time increases together with the complexity of an analysis being performed and the amount of computational resources being utilized. In a traditional HPC-based scientific computing centre responsibilities for detecting and handling errors are divided between the end user who is responsible for errors that occur in their data or algorithms, as well as their encoding of cluster compute jobs, and the cluster IT personnel who are responsible for any errors that are caused by the overall software and hardware infrastructure failures.

Although, it appears as if this division of responsibilities favours the end user by relieving them of the need to handle infrastructure issues, it has a notable downside. The underlying infrastructure and run-time environment are opaque to the end user who is only free to submit compute jobs and collect their results. On the other hand, the majority of errors (of any kind) manifest themselves as compute job failures and it is up to the end user to discern which of these are problems of their algorithm and which are problems of the infrastructure. Because of the opacity of the runtime environment, the end user has few tools at their disposal to be able to accomplish this task effectively and can spend significant effort troubleshooting issues that are outside of their domain of responsibility before being able to hand the incident resolution over to IT.

By contrast, in a cloud computing environment the end user has ownership of the health of the entire virtualized system including the infrastructure and the workloads that are running on it. This alleviates the issue of environment opacity described above but places a slew of new requirements on the operator of such a system when it comes to management of error conditions. Methods for detection and handling of errors differ depending on the source of the errors and we describe these in further detail below.

In general the following sources of error are identifiable and require appropriate handling:

- Errors within underlying scientific algorithms or the data they operate on
- Errors within the workflow definition
- Errors within the workflow engine
- Errors within the virtual infrastructure
- Errors within the bare metal hardware/software infrastructure and virtualization layers

Errors Within Algorithms or Data

Most scientific software is experimental by its nature and is developed by relatively few individuals compared to industry software, thus it is reasonable to expect that error conditions should occur within such software with a higher frequency than within mature and stable enterprise software. There are two standard mechanisms by which a program can return error conditions (whether they arise due to an error within the program or bad data) – return value, and log files. Both of these should be collected to have the most accurate representation of the state of running algorithms.

Additionally, multiple algorithms running on the same virtual machine will compete for its limited resources and sometimes, despite best resource planning efforts, will deplete them, with adverse effects on system performance. For instance, a system that runs out of physical memory may start using virtual memory and causing excessive memory swapping, thus severely degrading performance. Although some of the information necessary to diagnose such conditions may be available via system logs, it is definitely not comprehensive, and will typically allow the operator to deal with issues in a reactive rather than proactive manner. It is desirable, instead, to be able to actively monitor the trends in resource utilization within the cloud computing cluster and deal with potential resource bottlenecks before they arise.

The most typical resolution strategy for errors of this type would be to fix the underlying algorithm and re-run it for all, or only the affected samples under study.

Errors Within the Workflow Definition

When creating workflows it is possible to introduce errors that will only manifest themselves at runtime. Such errors will typically either cause a workflow task to fail, thus failing the entire workflow, or will cause a workflow task to stall, preventing further progress. The workflow engine needs to collect information about all failures and present it to the end user on a management interface, as well as recording it in the engine log files. Information about typical task runtimes should also be recorded to aid identification of tasks that have stalled.

Such errors would typically be addressed by fixing the workflow definition and issuing a new version of the workflow that will need to be re-run on all samples.

Errors Within Workflow Engine

As the workflow engine consists of a large number of running programs, any number of these may encounter error conditions during their operation. Each program should have a log file where such information can be gathered.

These errors would typically be addressed by patching the workflow engine or possibly restarting services that got into a bad state and the workflows that are mid-flight need to be resilient to such a situation.

Errors Within Virtualized Infrastructure

A cloud computing cluster may consist of hundreds of Virtual Machines, each machine in turn running hundreds of programs simultaneously. Given the large size of the computational fleet it is not uncommon for entire Virtual Machines or significant components thereof to fail, either by issuing an error signal or simply by becoming unresponsive or unreachable on the network. A key source of information about error conditions at the machine level is the operating system log file and it should provide the necessary diagnostic information when an error signal is present. For the cases when this signal is not present, however, and a VM simply stops responding a more active monitoring system is required – one that will periodically communicate with a Virtual Machine and collect its vital stats.

Errors within the VM Infrastructure are typically resolved by either patching and restarting services on the affected VM or by terminating and recreating the VM from scratch. The workflow system needs to be robust to both of these resolution strategies requiring only a minimal amount of work to be redone.

Errors Within Bare Metal Hardware/Software

As all cloud computing clusters represent virtualized hardware that is running on some bare metal server in a particular data center it is sometimes the case that the underlying hardware or software fails, thus rendering the Virtual Machine unusable. It is usually not possible for a cloud end user to gain visibility into the bare metal layer, and the responsibility for detecting and handling such issues generally falls on the cloud operator. From the user perspective the VM simply fails or stalls, and although the user requires methods for detecting such conditions, the resolution strategy is typically to recreate the VM again. The size of the data center for a typical cloud provider is such that if the underlying issue only affects one or a small number of bare metal servers, the probability of the new VM being scheduled on an affected server for the second time is quite low and, thus, computation may resume normally. When large scale network or other hardware issues affect the entire data center it may be necessary for the user to tear down and recreate the entire cloud computing cluster.

Depending on the magnitude of the issue in question, the resolution strategy may involve recreating individual VMs or the teardown and re-creation of the entire workflow system. In order to avoid significant data loss in this case effective data backup and recovery strategies are required on behalf of the workflow system operator.

Based on examining the typical sources of errors at various layers of the system above, the following error detection and mitigation mechanisms can be identified:

- System Monitoring
- Management Interfaces
- Log Files
- Self-Healing

System Monitoring

As previously noted, error conditions often do not occur spontaneously but instead are a result of contention for finite resources by various programs, or a byproduct of events outside the scope of the workflow system itself. Moreover, error conditions do not always result in program crashes that can be recorded to a log, but can instead cause a system to stall, become unresponsive, or unreachable on the network. If the underlying cause of the issue is identified in time before the system reaches a critical state it is at times possible to gracefully recover from the situation without reaching a crash and with minimal loss of work or productivity. To make this detection possible an active monitoring system is required to be deployed on the cloud computing cluster. The job of such a monitoring system is to keep track of all Virtual Machines that are part of the cluster and collect monitoring metrics indicative of the health of each.

The following set of key metrics need to be supported:

- CPU load
- Memory load
- Free memory
- Page faults
- Swap size
- Free disk
- Disk latency
- Disk throughput
- Disk IOPS
- Open files
- Network latency
- Network throughput
- Number of open sockets
- Number of database clients
- DB Transactions per second
- Transaction rollbacks
- DB number of connections
- DB Error count
- HTTP errors
- Queue size
- Queue spillover

In order to be able to detect minute conditions that adversely affect the health of the cluster the monitoring system needs to sample all of the above metrics from each VM with a sub-second frequency. Furthermore, to enable detection of trends in system health, the monitoring system needs to retain collected data over a period of time that is as long, or longer, than a typical workflow execution time (which in the case of scientific workflows can be days or even weeks). Since a Virtual Machine that is experiencing an error condition may become sluggish or entirely unresponsive it is instrumental that the metrics data that is being collected for each machine is quickly shipped off to another machine that can house and aggregate all such data across the cluster.

In order to allow end users to make use of the collected metrics for decision making the monitoring system needs to have graphing capabilities so that evolving trends in system health are made most evident. A set of graphical dashboards should be available to the user that demonstrate current and past cluster state based on the metrics above and a configurable time horizon. These dashboards will then be used during the course of system operation to identify potential issues and guide preventative or mitigative measures.

Monitoring Alarms

Because actively monitoring cluster state for potential issues via a set of dashboards is a time consuming task that may be prone to errors on behalf of the human observer an additional layer of the monitoring system should provide automatic notifications to the end user when the system enters a dangerous state such as high memory usage or CPU thrashing.

The user should be able to define a set of rules that express the conditions that are indicative of a system issue and require human intervention. Each rule should specify a metric or set of metrics, a set of thresholds, and an action. The monitoring system should continuously evaluate the metrics specified in each rule against the specified threshold, and when the metric value breaches the threshold the system should raise an alarm with the end user via the specified action.

Because large scale events like network outages may cause many metrics to breach their stated thresholds at the same time the Alarm System should aggregate all similar events into a single event, where possible, to avoid overwhelming the end user with notifications that all have the same root cause.

Management Interfaces

Keeping track of the various moving pieces of a distributed workflow system requires a set of management interfaces so that the user can get an overview of overall system state and maintain control of the system. Such interfaces should show all of the Virtual Machines that are part of the cluster, what capabilities each machine has, what workflows are currently active, and state of any databases or queues that are part of the system. When error conditions occur within a particular sub-system, such as during the execution of a workflow task, these errors should be made visible on the corresponding management interface for that sub-system, along with any relevant details related to the error. Thus, a workflow management interface should show any failed tasks and provide remedial options to the user, directly on that interface, such as retrying or deleting a particular task or rescheduling the entire workflow. An interface that displays Virtual Machines should indicate when any of the VMs become unreachable or unresponsive and allow the user to delete and recreate the VM.

Log Files

As is evident from previous sections that describe possible sources of errors in the system, one of the most frequent mechanisms for recording error conditions is writing to a log file. While every Virtual Machine has a system-wide log file that contains error messages from many applications it is also customary for most applications to write their own log files. Since the workflow system consists of many components, many log files will get generated. Making use of the bulk of log data is typically produced by a large size cloud computing cluster is a major problem.

In order to make the log data usable to the system operator it is required to harvest and aggregate all relevant logs from each VM that is part of the compute cluster. Since a VM may become unresponsive due to an error condition and the information on this condition may reside in a log file, it is necessary to run an agent on all VMs that will periodically ship the logs to another location for aggregation. The storage location for logs should have substantial capacity, as a large scale cloud computing cluster can generate many TB of logs per day.

The logs need to be parsed according to their format and information relevant to any conditions of interest should be extracted, along with necessary metadata, such as the host IP address and timestamp. The parsed output should then be aggregated by error condition to produce summary, as well as detailed, reports to present to the user in the form of dashboards. Parsed log information should be indexed and retained for further querying when a user decides to investigate a particular issue they discovered via the dashboard.

Self-Healing

Although comprehensive monitoring capabilities make it easier for humans to identify when and where error conditions occur, the cost of the human intervention required to interpret and act upon the monitoring data scales with the number of VMs employed on a project and can thus constitute a large component of a project's operating costs. In order to mitigate these costs and allow the efficient operation of large virtual clusters with minimal human intervention it is desirable to have a level of automation within the error detection and remediation systems such that adverse conditions that affect the cluster can be automatically detected and resolved by the system only alerting the human operator when automated action is impossible or fruitless.

The self-healing capabilities of the system should thus include the ability for the user to define a set of rules that specify normal and exceptional operating conditions of the system with respect to a predefined set of metrics such that issues can be identified in an automated fashion. This functionality should exist at multiple levels of granularity to facilitate the detection of a variety of issues, such as:

Infrastructure issues - Issues that affect the underlying VMs and can be detected via low level metrics related to high CPU utilization, memory, network bandwidth consumption, etc.

Supporting Services issues - Issues that affect the services of the framework itself, such as databases, queues, etc., detected via metrics on the individual services.

Payload issues - Issues that affect actual scientific payloads that are being executed and signal defects within the algorithms that are being executed. These require instrumentation of specific metrics for each type of algorithm that is being run in order to capture the requisite performance details.

A mechanism should exist that can perform corrective actions in an automated fashion when monitoring rule conditions are breached. This mechanism should be capable of facilitating a variety of repair tasks, such as rebalancing work onto different hosts, restarting failed services, adding hard-drive space, or destroying and rebuilding entire VMs. When automated remedial action is not possible or does not succeed the system should be able to notify human operators in a proactive manner.

3.2 Non-functional Requirements

Alongside the functional requirements for the system that have been detailed in the previous section there are a number of non-functional requirements that need to be considered. Chief among these are:

- Scalability
- Availability
- Ease-of-use
- Interoperability

3.2.1 Scalability

A major reason for utilizing cloud technologies for scientific computing is the need to compute over large data sets, ones that would otherwise be impractical or extremely costly to compute over. Thus building a system that can scale with the demand for computational resources is one of the key non-functional requirements for a successful scientific workflow framework. It is worth noting, that given cloud computing's prevalent cost model of charging for usage by the hour or minute it is equally important to be able to scale the system in both directions i.e. up when about to launch a massive compute job, and down when the major computation is finished, in order to operate the system in a cost-effective manner.

For purposes of scalability the workflow framework can be thought of as consisting of two types of components, which scale due to different factors:

Worker VMs - These Virtual Machines are the computers that are responsible for running actual scientific algorithms that are encoded within workflow tasks. Different configurations of worker VMs may be required depending on what type of algorithm is invoked - ones with more RAM, CPU, disk, etc. The number of worker VMs required scales with the resource demands of each workflow and the number of samples that are under analysis. The majority of VMs on the cluster will be of this type. When no active workflows are present all worker VMs can be destroyed. Creating new worker VMs when the need arises should also be very easy within the workflow system.

Control VMs - Control VMs are Virtual Machines that are responsible for housing the various control components of the workflow system such as the Workflow Scheduler, Metrics Store, Management UIs, etc. Depending on the function served by each Control VM it can scale based on the number of Worker VMs, granularity of workflow tasks, or number of users of the system. Although it should be possible to scale the number of Control VMs up and down depending on need, a baseline number of Control VMs will be running for the duration of the cluster lifetime.

Based on the different types of components that compose the workflow framework the following scaling use case can be identified and need to be handled:

- Bootstrap a new installation into a minimal working configuration
- Scale up worker fleet to meet anticipated resource demand increase
- Scale down worker fleet to match resource demand decline
- Scale up control VMs to meet anticipated resource demand increase
- Scale down control VMs
- Destroy entire compute cluster

Bootstrap New Installation

When installing on new clouds or Availability Zones it should be possible to easily install a basic working system with all of the necessary control and worker VMs to be able to handle some minimal amount of computational load.

Of key importance is the distinction between systems that scale horizontally and systems that scale vertically. For systems that scale horizontally, it is relatively easy to scale up by adding new machines. Systems that scale vertically, such as database servers, do not easily allow adding new machines and instead require upgrading the resources within a single machine in order to scale up. It is generally more challenging and time consuming to scale vertically than horizontally.

When specifying a minimal installation it is important to set up those machines that scale vertically to a size that is larger than what is required by minimal system usage in order to avoid having to perform frequent upgrades. All databases and data stores fall into this category because of the complexity of upgrading the underlying storage once that database is in service.

Scale Up Worker Fleet

When the workflow framework is required to be able to perform more computation per unit time than it currently can, it should be easy to scale up the worker fleet by any number of VMs without interrupting existing jobs or requiring code changes.

Scale Down Worker Fleet

When demand for computational resources wanes, it should be easy to destroy any number of idle worker VMs without interrupting presently running jobs or requiring code changes in order to minimize operating costs.

Scale Up Control VMs

Based on an increased number of worker VMs or increased number of users of the system it should be easy to scale up the Control VMs to maintain smooth operation of the cluster.

As before, it is important to delineate between vertically and horizontally scaling systems. For horizontally scaling systems, such as web-servers or distributed queues, scaling up may require bringing up a second or third server and setting up a load balancer. For vertically scaling systems, scaling up means migrating the existing system to a larger VM (with more memory, or cores), or migrating an existing data store to a larger storage device. Thus, as before, when scaling up vertically scalable systems, it is important to scale up appropriately to be able to handle all foreseeable, not just immediate, usage.

Scale Down Control VMs

Most Control VMs (for instance those serving data stores) tend to only grow in size, thus not ever requiring scaling down. It should be possible, however, to scale down certain components, such as web servers hosting management UIs, when a prolonged period of low activity is anticipated.

Destroy Cluster

When a long period of downtime is expected it may be required to destroy all of the active VMs in order to avoid paying for underused resources. A process should exist for detaching and conserving all of the active data stores, so that they can be reattached during the next bootstrap. All the appropriate configurations that are necessary for bootstrap should also be retained. At that point it should be safe to terminate all active VMs.

3.2.2 Availability

Availability refers to the definition of the circumstances under which the system as a whole, or its components, are able to perform their intended duties.

Although a scientific workflow system is an important computational aide it typically does not constitute a life-critical device that has extremely high availability requirements, such as those that might be found in a hospital life-support, or aeronautical system. Nevertheless, it is reasonable to expect that under normal operation, the system will be functional on a 24 hour, 7 days a week basis, and this intended schedule should inform system design.

Additionally, the following concerns regarding Availability should be considered:

- Service Redundancy
- Service Upgrades
- Backup and Disaster Recovery

Service Redundancy

Some components of the workflow system are such that if they crash or go offline, the system will continue functioning, although, perhaps, in a reduced capacity - Metrics Monitoring, and Log Aggregation are examples of such components. Other components, like Workflow Scheduler, Workflow State Management, and others, were they to go down, would bring down the entire operation. Thus, in order to maintain service availability during adverse events, it is important to build service redundancy into the design of the system.

Typically, Service Redundancy takes the shape of having a second standby server that is able to stand in place of the main server if it was to go offline. Multiple levels of standby are available, and should be used as appropriate:

Cold Standby - Set up the backup machine when primary machine fails.

Warm Standby - Backup machine is set up and will be powered on when primary machine fails.

Hot Standby - Backup machine is set up and powered on, traffic will be diverted to it when primary machine fails.

Both Active - Both primary and backup are actively serving traffic in a load balanced manner.

Service Upgrades

Upgrading software is the leading cause of planned service outages. From a customer perspective, although the disruption is planned, and warning can be given, it is still a disruption and thus should be avoided where possible.

It is not always possible to upgrade software without a service interruption, but utilizing standby servers and appropriate load balancers can increase the number of deployments that are performed without a disruption to the users. Thus, service upgrades should be scripted such that a backup server is brought online, to which active traffic will be diverted by the load balancer while the main service is upgraded. Once the main server is upgraded, the load balancer should switch traffic back and allow the backup server to be upgraded, and brought back offline where appropriate.

Backup and Disaster Recovery

In the event of a catastrophic system failure it should be possible to recover as much of the data as is warranted by the scientific use case the system is being utilized for, and resume normal operation as soon as possible. Data backup and recovery procedures should be put in place for all of the main data stores, especially those housing workflow scheduling and state management facilities as loss of these may potentially mean not only redoing the in-flight analyses but also past ones, as otherwise reproducibility would be lost.

3.2.3 Ease-of-use

Cloud computing is a relatively new technology and coupled with the complexity of a distributed workflow system can create a learning curve that would hinder system adoption. To help potential users through the learning curve and encourage a wider adoption of the product the following measures are required:

- Detailed User Guide
- Detailed code comments
- Standardized code style
- Hands-off operation where possible
- Contextual help
- Intuitive Management UI

3.2.4 Interoperability

Because data needs to be fed into it and results need to be retrieved out of it, a cloud based distributed workflow system does not exist in isolation from the rest of the world. Thus, a level of interoperability is required whereby systems upstream and downstream from the workflow framework need to be able to successfully interact and exchange information with the framework.

The most common need for interaction is when feeding data into the system. Since sample management is a complex issue on its own it falls outside of the scope of requirements for the workflow framework and typically resides within the realm of Laboratory Information Management Systems or Data Repositories. Yet, samples needs to be made available to the framework in order to enable computation. Thus, the framework needs an interface whereby locations of samples under analysis can be specified.

Another use case for interoperability is the desire to make the framework amenable to automation via scripting, so that, for instance, new samples that come into the

laboratory can be automatically scheduler for a battery of standardized workflows to be performed, or a new algorithm version becoming available will automatically trigger a re-running of all samples under study using the new tool.

In order to enable this level of interoperability a set of APIs must be developed that provide all of the main framework operations through an interface that another program can interact with.

User Interfaces developed for the framework should utilize the same set of APIs where possible for consistency purposes, and to ensure that API code is frequently executed.

3.3 General Design Principles

To address the need for a large-scale cloud-based distributed workflow system that is suitable to scientific computing applications we present the design of our framework called Butler - a software toolkit built to implement the requirements specified in the Requirements chapter of this document.

Although a detailed description of system Architecture and Design follows, we begin by describing several guiding principles that have been adopted in the design of this system:

- Existing Open-Source Software
- Service Orientation
- Cloud Agnostic
- Open-Source License

3.3.1 Existing Open-Source Software

The scope of the requirements for a workflow system of the nature described in this work are quite vast and building such a system from scratch would take years of effort from an entire software team. On the other hand, many of the requirements of the system can be readily met via existing software products. Although commercial software products tend to have better technical support, in the interest of cost savings, and in order to keep the entire solution open source we have opted to use all Open Source Software components when building Butler.

Since keeping the amount of new code that needed to be written to build Butler to a minimum was one of the cornerstones of system design, a very substantial portion of the overall system consists of 3rd party OSS frameworks that are integrated together to produce Butler. These include:

- Hashicorp Terraform[211] (<https://github.com/hashicorp/terraform>) - for Cluster Lifecycle Management
- Hashicorp Consul[47] (<https://github.com/hashicorp/consul>) - for Service Discovery and Service Health Checking
- Saltstack[187] (<https://github.com/saltstack/salt>) - for Cluster Configuration Management
- Apache Airflow[17] (<https://github.com/apache/incubator-airflow>) - for Workflow Management
- RabbitMQ[176] (<https://github.com/rabbitmq/rabbitmq-server>) - for Queuing
- Celery[96] (<https://github.com/celery/celery>) - for Task Scheduling
- Collectd[203] (<https://github.com/collectd/collectd>) - for Metrics Collection
- InfluxData InfluxDB[98] (<https://github.com/influxdata/influxdb>) - for Metrics Storage
- Grafana[163] (<https://github.com/grafana/grafana>) - for Metrics Dashboards
- Logstash[201] (<https://github.com/elastic/logstash>) - for Log Harvesting
- Elasticsearch[19] (<https://github.com/elastic/elasticsearch>) - for Log Indexing and Aggregation
- Kibana[106] (<https://github.com/elastic/kibana>) - for Log Event Dashboards

These products were selected based on their ability to fulfill the specified requirements as well as their overall viability as Open Source projects. Viability was generally evaluated based on the following set of criteria:

- Number of Github stars
- Number of repository contributors
- Number of commits in the past year

At the time of writing (September 2016), these metrics are evaluated as follows:

3.3.2 Service Orientation

One of the key requirements for Butler is Scalability i.e. the desire to be able to scale the amount of resources utilized by the framework up and down arbitrarily according to analysis needs. Applications that are monolithic in nature suffer from scalability issues due to the large number of competing constraints within application components. To help alleviate this concern we take a Service Oriented approach in the design of

Table 3.1: Open Source Framework Viability

Product	Number of Stars	Number of Contributors	Commits in last 12 months
Terraform	5611	638	4543
Consul	7220	250	1129
Saltstack	6850	1580	9769
Airflow	3419	178	1394
RabbitMQ	1964	63	1038
Celery	5224	462	854
Collectd	1246	262	1104
InfluxDB	8773	237	2299
Grafana	11666	320	2952
Logstash	6405	348	598
Elasticsearch	18285	691	6279
Kibana	5957	146	3409

the system. Butler is composed of a number of loosely coupled services each of which implements a particular function. Because the services are decoupled, each service can be optimized and scaled individually, according to user requirements. On the other hand, the complexity of the overall application is increased somewhat because of the need to deploy and manage separate services that are in communication with each other.

Another benefit of Service Orientation is the ability to independently upgrade components of the software without affecting other running components. As an example, the Collectd metrics collection component can be patched independently of the rest of the system, thus increasing system Availability.

3.3.3 Cloud Agnostic

Because of its early entrance on the cloud computing scene, the AWS Cloud by Amazon.com Inc currently enjoys a significant lead in this market segment, having over 31% of the entire public cloud-computing market:

Each of the top 4 cloud service providers - Amazon, Google, IBM , Microsoft, as well as smaller cloud providers that use the Openstack platform, provides not only the basic IaaS offering, but also an entire ecosystem of cloud based components - a PaaS, including networking, queues, databases, etc. Thus, it may be tempting to select one of these providers and build an entire software system that is based on a single vendor's offerings. This has the potential benefit of significantly simplifying system architecture and providing a single point of contact for troubleshooting.

It is, however, our opinion that taking such an approach would limit the appeal of the system to a wider user base. This opinion is driven by several considerations:

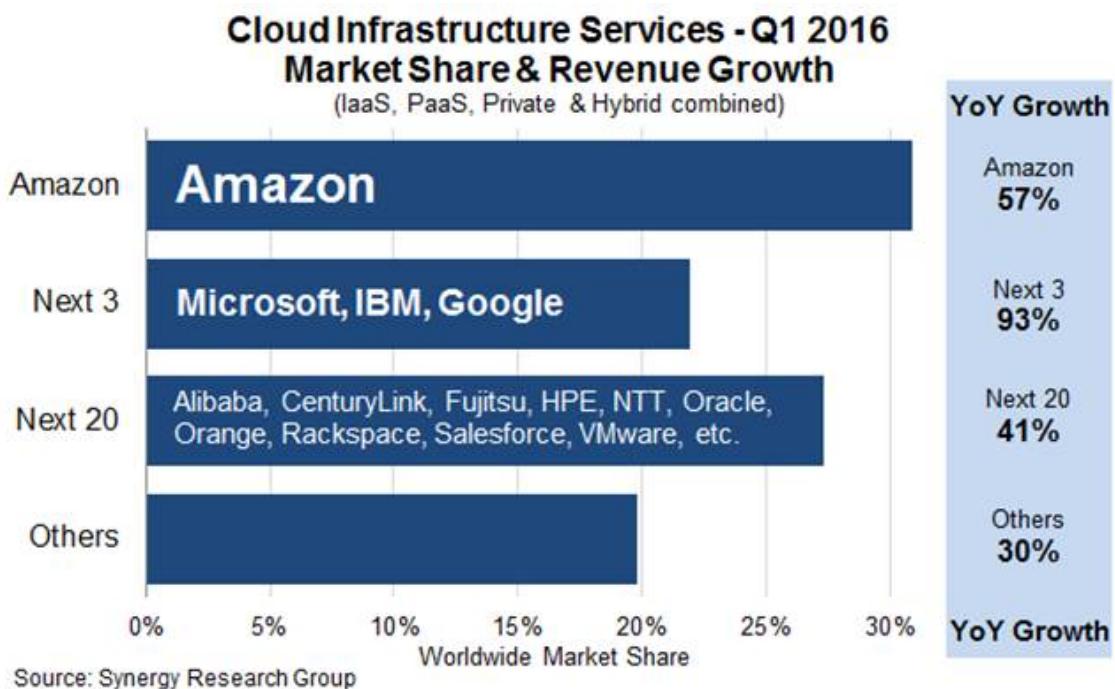


Figure 3.2: Cloud market share by vendor

- The cloud computing market segment enjoys a great deal of growth and significant shifts in growth year-over-year as evidenced by Figure 3.2, thus committing to a particular platform that is seen as a current market leader today, may limit the usability of the software when the chosen vendor falls out of the race in the future.
- Because the market segment is highly competitive, end users can benefit significantly from limited time deals offered to them by cloud providers if they are flexible about what platform to deploy on.
- Selecting one vendor induces vendor lock-in, possibly forcing adoption of inferior technologies to stay consistent with vendor choice.
- Public and Private clouds typically operate on different software stacks. The nature of the data that is subject to scientific analysis may dictate where the analysis is able to proceed.

On the other hand, supporting multiple cloud vendors has its own set of drawbacks:

- Handling multiple APIs for different vendors increases system complexity.
- A solution that is vendor agnostic may lack certain capabilities that are only available to a subset of the vendors.
- Some code duplication is inevitable when dealing with multiple platforms.

Based on considerations above we have taken the path of creating a cloud-agnostic system, i.e. one that will run on any major cloud providers, public, or private.

3.3.4 Open Source License

We adopt an open-source GPL v3.0 license for Butler.

3.3.5 Overall System Design

Overall, the Butler system can be thought of as being composed of four distinct sub-systems, each of which fulfills a number of requirements from Section 3.1. These sub-systems are:

Cluster Lifecycle Management - This sub-system deals with the task of creating and tearing down clusters on various clouds, including defining Virtual Machines, storage devices, network topology, and network security rules. It implements requirements from sections 3.1.1, and 3.1.2.

Cluster Configuration Management - This sub-system deals with configuration and software installation of all VMs in the cluster. It implements requirements from section 3.1.3.

Workflow System - The Workflow sub-system is responsible for allowing users to define and run scientific workflows on the cloud. This sub-system implements requirements from sections 3.1.4, 3.1.5, and 3.1.6.

Operational Management - This sub-system provides tools for ensuring continuous successful operation of the cluster, as well as for troubleshooting error conditions. It implements requirements stated in section 3.1.7

Each sub-system is described in full detail below.

3.4 Cluster Lifecycle Management

Before any computation can happen on the cloud a cluster of Virtual Machines is needed. The scope of Cluster Lifecycle Management includes:

- Defining hardware configuration for VMs
- Defining initial basic software configuration for VMs
- Defining storage devices
- Defining network topology
- Defining network security
- Creating and Tearing down VMs

Detailed requirements for these capabilities are specified in sections 3.1.1, and 3.1.2.

To fulfill these requirements in a cloud agnostic manner Butler utilizes a framework called Terraform, developed by Hashicorp.

3.4.1 Terraform

Terraform is an Open Source framework for cloud agnostic cluster lifecycle management, that has been built by Hashicorp Inc., a San Francisco, California based company, and is distributed via a Mozilla Public License. The source code for Terraform is hosted on Github at <https://github.com/hashicorp/terraform>, and at the time of this writing (September, 2016) the latest release of the software is version v0.7.3

Terraform uses a proprietary human and machine readable file format for specifying cluster configurations that is called HashiCorp Configuration Language (HCL). Using this language the end user can define a number of constructs for cluster management, most important among them are - providers, resources, and variables.

Terraform Providers

Terraform providers enable the framework to talk to different cloud provider APIs. Each provider is responsible for translating HCL configurations into cloud-specific API calls. At the time of this writing the following Providers are available:

- AWS
- CenturyLinkCloud
- CloudFlare
- CloudStack
- Cobbler
- Datadog
- DigitalOcean
- DNSimple
- Google Cloud
- Heroku
- Microsoft Azure
- OpenStack
- SoftLayer
- Scaleway
- Triton
- VMware vCloud Director
- VMware vSphere

Typically in order to use a particular provider the user needs to insert a provider block into their configuration file where they specify details relevant to communicating with the particular API in question, such as - endpoint URL, username, password, SSH keyname, API key, etc., as seen in Figure 3.3.

```
provider "aws" {
    access_key = "foo"
    secret_key = "bar"
    region     = "us-east-1"
}
```

Figure 3.3: Example Terraform provider configuration

Once the user has specified a provider they can declare provider-specific Resources that define their cluster.

Terraform Resources

Resources represent different objects such as VMs, network routers, security groups, disks, etc., that the user can create on a given cloud. Each resource has a set of configuration options that can be specified to customize its behaviour. An optional *count* attribute defines how many instances of the resource need to be created in the cluster.

```
resource "aws_instance" "web" {
    ami      = "ami-408c7f28"
    instance_type = "t1.micro"
}
```

Figure 3.4: Example Terraform AWS instance configuration

Most Terraform configuration involves configuring resources.

Terraform Variables

Terraform variables are similar to variables in any other programming context. They consist of values assigned to labels, that can then be used for lookup elsewhere. Variables can be of string, list, or map type.

```
variable "users" {
    type    = "list"
    default = ["admin", "ubuntu"]
}
```

Figure 3.5: Example Terraform variable definition

Users typically specify variables in a separate configuration file and then use them throughout their cluster definition.

One special case of using variables comes from specifying secret values such as passwords or secret keys that the user would not want to commit to a source repository. In this case, a variable can be referred to inside the configuration file, while being defined as an environment variable on the machine that Terraform will be executed on. The user prefixes the variable name with a special prefix – `TF__VAR__` which signals Terraform to parse the environment variable as a Terraform variable and allow appropriate substitution at runtime.

Terraform Provisioners

When a Virtual Machine is created the user may want to place certain files on it or run certain commands such as starting services or registering with a cluster manager, in order to bootstrap it. This purpose is served by Terraform Provisioners, which define code blocks that are executed on the target resource upon creation.

```
# Copies the file as the root user using SSH
provisioner "file" {
  source = "conf/myapp.conf"
  destination = "/etc/myapp.conf"
  connection {
    user = "root"
    password = "${var.root_password}"
  }
}

# Copies the file as the Administrator user using WinRM
provisioner "file" {
  source = "conf/myapp.conf"
  destination = "C:/App/myapp.conf"
  connection {
    type = "winrm"
    user = "Administrator"
    password = "${var.admin_password}"
  }
}
```

Figure 3.6: Example Terraform provisioner definition

Terraform Installation

Terraform is installed via a binary file downloaded from the Hashicorp website or by compiling the source code from github. It is a lightweight application that can be run from either the user's local machine, or from a special host on the target cloud environment. The application consists of a terraform CLI that the user can interact with by issuing shell commands. Typically users will combine their Terraform configuration files (stored in a source code repository) with a set of locally defined environment variables to set up and manage their clusters via the CLI.

Terraform Cluster Lifecycle

The key task of Terraform is to perform Create, Read, Update, and Delete on cluster resources. Create and Update operations are accomplished by issuing a `terraform apply` command at the shell, while the shell is pointing to a directory with Terraform resource definitions. If the resources specified in the configuration do not yet exist, they are created. If the resource definitions have been changed since the last time `terraform apply` was run, they will be brought into a state consistent with the latest definitions. This may involve updating existing resources where possible, or recreating them, where an update is not possible.

Terraform determines what changes need to be made in order to perform a successful Update via a file that is called a State file. This file specifies in a JSON format the current state of all infrastructure managed by Terraform. Running `terraform apply` causes the tool to inspect current state and compare it to the target state, issuing any necessary commands to update current state to the target.

The Read operation simply displays the current Terraform state file via the `terraform show` command.

The Delete operation is accomplished via the `terraform destroy` command.

Other commands allow the user to validate the syntax of their configuration files, perform a dry run of resource creation, manually mark resources for recreation, and others.

3.4.2 Terraform Use in Butler

Butler comes with a set of Terraform configuration files that define templates for all of the VMs that constitute a functional Butler cluster, as well as configurations for network security. As previously stated a Butler cluster consists of Control VMs and Worker VMs - templates for both are available. The users are expected to adapt the templates as needed for their use case, providing their own credentials, cluster size, and other configurations.

Example Configurations

Listing 14 demonstrates the Butler configuration file used to create 175 identical worker VMs that differ only by their hostname.

The provider definition shows the procedure for setting up an OpenStack provider as well as demonstrating usage of variables where `user_name`, `tenant_name`, and `auth_url` are expected to come from a separate variable definition file, and `password` is expected to come from an environment variable.

The resource section shows definition of an OpenStack specific VM type `openstack_compute_inst`

which has attributes like `image_id`, `flavor_name`, `security_groups`, `network`, etc. The `connection` definition within the resource specifies how users will be able to connect to the newly created VMs. In this case it is accomplished via SSH using passwordless key-based authentication via a pass-through bastion host on the cloud.

Of further interest is the mechanism by which the creation of multiple instances of the same type is accomplished. The resource definition admits a `count` attribute which specifies how many instances need to be created. Furthermore, a `count.index` property keeps track of which instance is being created at run-time and can be used to provide unique hostnames to each instance as follows - `name = "\${concat("worker-", count.index)}`

Lastly, the `provisioner` section runs a set of commands that provide initial configuration for the new host upon first bootup. These include installing and running the Saltstack service which is used for configuration management, setting up machine roles that determine what capabilities this VM will have in the cluster, and telling the VM what the IP address of the cluster manager is.

Listing 15 demonstrates the definition of a security group under OpenStack. VMs that are put into this security group will have two network security rules applied to them - opening port 22 for SSH communication between hosts, and opening ports 4505-4506 to enable Saltstack communication.

3.5 Cluster Configuration Management

Although a Cluster Lifecycle Management system like Terraform can create a Virtual Machine using a machine image, and even run some initial configuration commands, it is not enough to successfully manage the configuration of an entire large-scale computational cluster. Machines in the cluster will have hundreds of programs installed and configured on them, oftentimes with intricate interdependencies, and inter-machine communication requirements. Moreover, different operating systems will typically have different commands and mechanisms for installing and configuring software, and it would be unnecessarily limiting to require the end user to commit to a particular flavour of operating system. To help accomplish these tasks, as well as detailed requirements from Section 3.1.3 we need to enlist the help of a Cluster Configuration Management system.

Several open source Configuration Management systems are available on the market today, the main options are:

- Chef
- Puppet
- Ansible
- Saltstack

Each system has benefits and drawbacks and a dedicated user base. Table 3.2 compares the github codebases for these products in terms of number of stars, number of contributors, and number of commits in the last year.

Table 3.2: Configuration Management Frameworks github summary

Product	Number of Stars	Number of Contributors	Commits in last 12 months
Chef	4417	474	1861
Puppet	4151	431	1474
Ansible	18720	1469	3483
Saltstack	6850	1580	9769

As can be seen from the data, all four are fairly active and stable projects, Ansible appears to be the most popular tool, and Saltstack is most actively developed, based on number of commits and contributors. Both Puppet and, Chef come from the first generation of configuration management tools having been initially released in 2005 and 2009 respectively, and suffering somewhat from having been trailblazers in the field. The largest complaint against both systems has been their unnecessary complexity and steep learning curve. Ansible and Saltstack, on the other hand, can be thought of as the second generation of configuration management systems, first released in 2012 and 2011, respectively. Both are based on simple to read and understand YAML-based configuration files, and have generally enjoyed greater adoption in the field.

For Butler we selected Saltstack to fulfill configuration management duties. The chief reason for selecting Saltstack over Ansible was that Saltstack appears to perform better when managing large clusters, whereas Ansible is known to suffer from increased lag in these scenarios. Since we anticipate to operate Butler clusters with several hundred VMs at a time we settled our choice on Saltstack.

3.5.1 Saltstack

Saltstack is an open source product that has been developed specifically for large scale configuration management. The key paradigm that Saltstack implements is declarative configuration management. This means that the user specifies declaratively, in a configuration file, what state a particular Virtual Machine should be in (in terms of installed and running software), and the Saltstack engine automatically compares the desired state to the actual state and carries out the necessary actions to match the two. As an added benefit, it does so in an operating system agnostic manner. In contrast to scripts that operate in an imperative manner via statements like `yum install apache` or `service httpd start`, Saltstack files describe a desired state with statements like `service.running` and `package.installed`. In the first case, the script would try to install the package a second time, even if it was

present, whereas Saltstack first figures out whether the package is installed and only installs it if it is missing.

Saltstack Architecture

The Saltstack architecture consists of a cluster of Minions that are managed by one or many Masters. A Master is a Virtual Machine that acts as the authority on configuration definitions within the cluster and issues commands that the Minions run. A Master needs to have configuration definitions stored locally on its disk or be available through a git repository. It runs a special salt-master daemon, and requires certain network ports to be open for communication.

Minions need to know how to find the master on the network (by IP address). Each Minion generates a unique key and presents it to the Master. Once a Master accepts the Minion's key there is a handshake and the Minion falls under the Master's control. The Minion runs a salt-minion daemon.

Each Minion can have a number of roles assigned to it and the Master maintains mappings between roles and configurations. Once the Master has determined what roles a Minion has it can issue the necessary commands to apply relevant configurations to the Minion.

Saltstack Data Model

The Saltstack Data Model has four main concepts - State, Pillar, Grain, and Mine. We consider each in turn.

A Salt State is simply the definition for what state some piece of infrastructure should be in. For instance, if we want some server in our cluster to be in the state of running a PostgreSQL database we need to do the following:

1. Create a postgres user
2. Create a postgres directory
3. Download the postgres-server package
4. Install the postgres-server package
5. Initialize the database
6. Override default configuration settings
7. Start the server

The corresponding Salt state that accomplishes the same task looks as follows:

Listing 1: Salt state for setting up a PostgreSQL server.

```
1  install_server:
2      pkg.installed:
3          - name: postgresql95-server.x86_64
4
5  initialize_db:
6      cmd.run:
7          - name: /usr/pgsql-9.5/bin/postgresql95-setup initdb
8          - unless: stat /var/lib/pgsql/9.5/data/postgresql.conf
9
10 /var/lib/pgsql/9.5/data/postgresql.conf:
11     file.managed:
12         - source: salt://postgres/config/postgresql.conf
13         - user: postgres
14         - group: postgres
15         - mode: 600
16         - makedirs: True
17
18
19 start_server:
20     service.running:
21         - name: postgresql-9.5
22         - watch:
23             - file: /var/lib/pgsql/9.5/data/*
```

The code for a Salt state is placed in a special file called an *.sls* file. All of the state definitions that the system knows about are arranged into a folder hierarchy where the name of each folder defines the name of the state. The state definition is then located inside the folder in a file named *init.sls*, as demonstrated in Figure 3.7 for the Airflow Workflow engine.

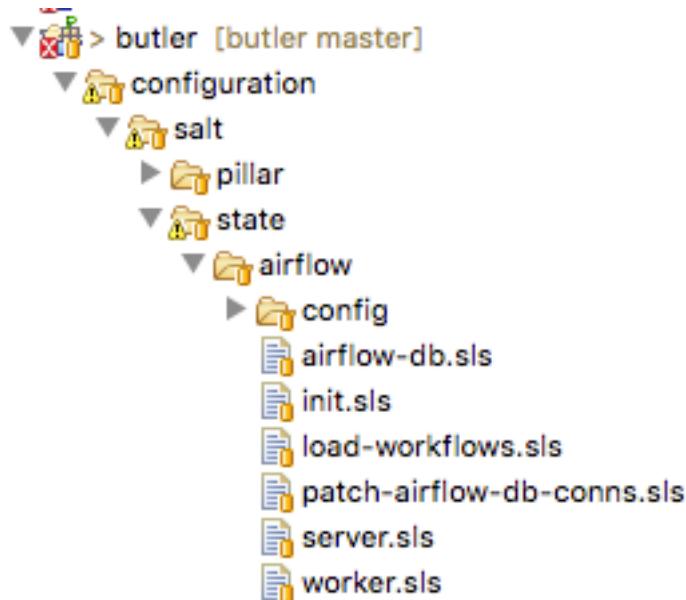


Figure 3.7: Salt state files for Airflow Workflow Engine

Several related states (such as those describing different installations of the same program) can be grouped together under the same parent state. Then each sub-state is placed into its own *.sls* file under the main state's folder, with the name of the file giving rise to that state's name. Figure 3.7 provides an example of this scenario where in addition to the main *airflow* state there are sub-states such as *airflow.server*, *airflow.worker*, *airflow.load-workflows* etc. Note that sub-states are referenced via *name_of_parent_state.name_of_substate*.

A Salt Pillar is a set of key-value pairs that are stored encrypted on a Minion and constitute look-up values that are relevant for that Minion's configuration. Examples of Pillar values can be usernames and passwords, locations of certain files, etc. A State definition can refer to Pillar values when configuring a system, and two identical VMs that differ only by their Pillar values will be parametrized differently at configuration time. One example of this is setting up the same server in a QA environment vs. Production. In QA the server may point to a test data directory with especially constructed data files, for testing purposes, whereas in Production the server would point to the actual data directory with real samples.

The Pillar are organized similar to States in a folder hierarchy of *.sls* files. Figure 3.8

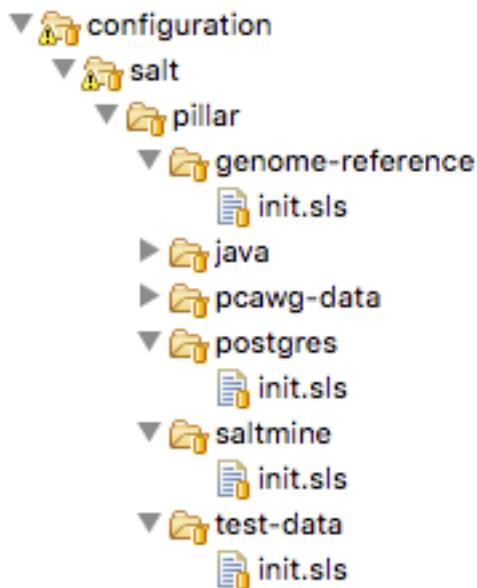


Figure 3.8: A set of Salt Pillar definitions

Listing 16 shows an example Pillar definition where information related to finding test data is stored.

Salt Grains are bits of information Salt collects about Minion state or characteristics. They include things like:

- Minion IP address
- Amount of RAM on minions
- Minion hostname
- Minion network interfaces

and others. The Grains can be used to introspect and pass on configuration values (like IP address) that are not known in advance. One of the most important uses of Grains is the ability to assign roles to a Minion via the Grains mechanism. Since roles define what states are eventually applied, adding or removing a role to a VM via Grains has a very significant side-effect.

The Salt Mine is a centralized repository of information about the state of all Minions that is stored on the Master. Information is passed into the Mine from Grains and other sources. It can then be used inside state definitions to further customize the system.

Listing 17 demonstrates how the Jinja templating engine is used to look up the IP Address of servers in the cluster that have the `consul-server` or `consul-bootstrap` role. Then this IP Address is used inside a State definition to join a cluster of similar machines. Without the Mine, this particular Minion would not know who to ask for this IP Address, but because the Mine is centralized on the Salt Master host this lookup is possible.

The Top File is the mechanism used in Saltstack to specify what VMs will have what States applied to them. The Top File provides a lot of flexibility in terms of how to accomplish this mapping. Mappings can be accomplished via hostname or any Grains values, and it allows regular expressions. The most flexible and, thus preferred, method of mapping States to VMs is via Roles.

Listing 18 demonstrates how the State mapping to Roles is accomplished in a Top File. Based on this Top File all VMs will get the `consul`, `dnsmasq`, and `collectd` states. VMs with the `monitoring-server` role will get `influxdb`, and `grafana`, and VMs with the `job-queue` role will get the `rabbitmq` State.

Controlling Saltstack

Control over the cluster is exercised from the Salt Master. The user establishes a shell session on the Salt Master and issues commands via the Saltstack CLI. Each command has the following syntax:

"`salt target_expression command_expression`" where:
`salt` is the name of the Salt CLI.

`target_expression` is an expression that determines what VMs to apply the command to. It can be a logical expression that combines hostnames, grains, and regular expressions.

`command_expression` is an expression that determines what actual command to run on the targeted VMs. The `command_expression` can be as simple as running a shell command on the target VMs, or it can apply a particular named state via the `state.apply` command, or it can apply all matching states via the special `state.highstate` command.

For example, `salt -G 'roles:worker' state.apply airflow.patch-airflow-db-conns` applies the `airflow.patch-airflow-db-conns` state to all VMs that have the `worker` role.

3.5.2 Saltstack Use in Butler

Butler uses Saltstack extensively in order to install software on the cluster. This includes software that is required to run Butler itself, as well as installing scientific algorithms required for running actual workflows on Worker VMs (as specified in Section 3.1.3 of the Requirements chapter). As seen in Figure 3.9 the Saltstack configuration in Butler consists of a set of State and Pillar definitions along with the Top Files that map these States and Pillar to various VMs in the cluster. These definitions are enough to configure a completely functional Butler cluster from a single shell command.

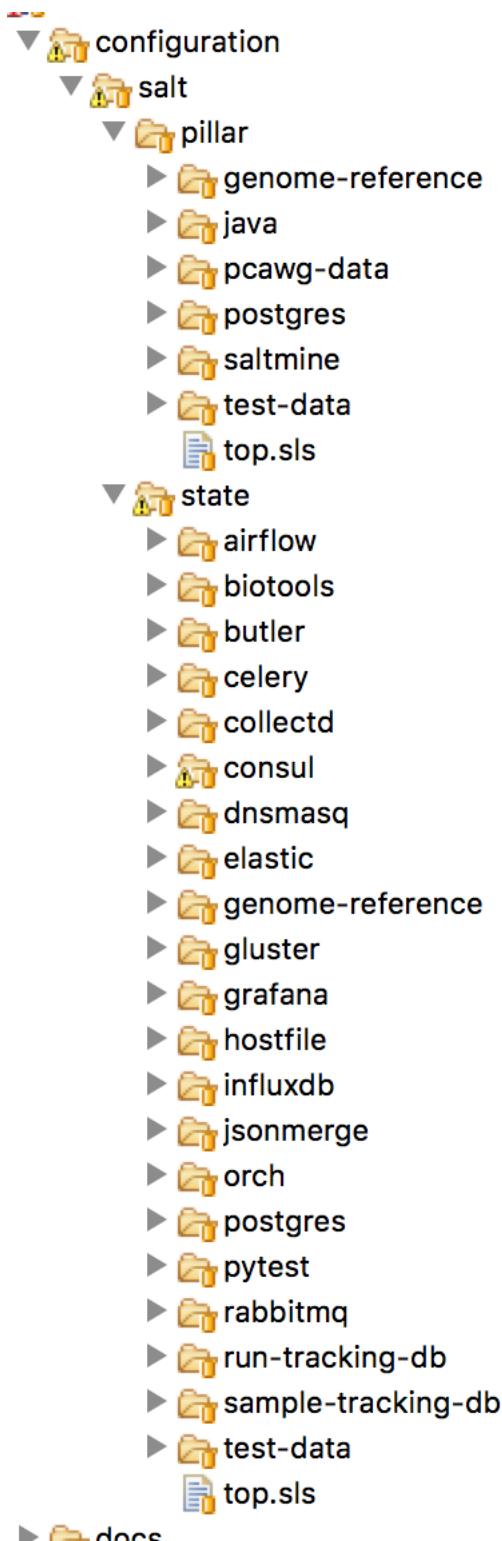


Figure 3.9: Salt States and Pillar used in Butler

A typical Butler installation that can support a cluster of up to 1500 CPUs consists of four Control VMs in addition to the Worker VMs, each has a separate Terraform profile. The Control VMs are:

salt-master - This machine is the configuration master node. Because this workload is typically only heavy during cluster setup, the same VM also acts as the Monitoring Server during regular operation.

db-server - This VM hosts all the databases that Butler uses.

job-queue - This VM hosts a Queue for distributed task processing.

tracker - This VM hosts all of the workflow engine components, as well as Analysis Tracking.

All of the VMs in the cluster get the following basic configurations mapped in the Top File:

```
'*':
  - consul
  - dnsmasq
  - collectd
  - elastic.filebeat
  - elastic.packetbeat
```

consul - A framework used to Service Discovery which will be described in detail in Section 3.7.5

dnsmasq - A local DNS server, to enable name lookups.

collectd - A Metrics collection agent.

elastic.filebeat - A server log harvester.

elastic.packetbeat - A network event log harvester.

Setting up the Salt Master

The first order of business when setting up a new Butler cluster is to bootstrap the Salt Master VM, as this VM is responsible for configuring and installing the software of all the other machines, including itself.

A Butler VM is typically provisioned from a base VM image, which has little more than the barebones OS, using Terraform. In the case of the Salt Master, the salt-master daemon is installed via Terraform's `remote-exec` provisioner. Salt's `highstate` command is then executed on the master itself in order to fully initialize it. At that point the Salt Master is ready to configure other machines that are part of the cluster.

As previously mentioned, because the load on the Salt Master is typically only high during initial cluster setup and during short bursts during normal operation, the Salt Master VM typically has another Saltstack Role mapped to it - that of the Monitoring Server. This role installs monitoring components that will be described in detail in Section 3.7

Setting up Other Butler Control VMs

The DB Server VM has a db-server Role mapped to it. Because databases are resource intensive software that does not scale horizontally, this VM does not have other roles within the cluster.

```
'G@roles:db-server':  
    - postgres  
    - run-tracking-db  
    - grafana.createdb  
    - airflow.airflow-db  
    - sample-tracking-db
```

The Top File mapping of States to the **db-server** role ensures that the PostgreSQL DB Server is installed as well as a number of databases that are used by Butler for tracking scientific analyses, workflow statuses, analysis samples, and performance metrics.

The Job Queue VM has a **rabbitmq** state mapped to it in the Top File, to install the RabbitMQ queueing system.

The Tracker VM correspondingly has a **tracker** role and the following state mappings:

```
'G@roles:tracker':  
    - airflow  
    - airflow.load-workflows  
    - airflow.server  
    - jsonmerge  
    - butler
```

These states install and configure the Airflow Workflow engine, load available workflows, and check out and install the Butler codebase from github. The codebase is needed to run the Butler CLI which is used to set up and manage Butler analyses. Thus, most interactions the users have with Butler occur from the Tracker VM via the Butler CLI.

Setting up Butler Workers

While Control VMs will be quite similar from one installation of Butler to the next, the Worker VMs will differ quite a bit, depending on what types of analyses are anticipated to be performed. The base Worker VM has the **worker** role which simply allows such VMs to run workflows in principle by installing the necessary components of the workflow engine and Butler Analysis Tracker.

```
'G@roles:worker':  
    - dnsmasq.gnos  
    - celery  
    - airflow  
    - airflow.load-workflows  
    - airflow.worker  
    - butler
```

The actual scientific algorithms that are required for running particular analyses are installed onto Workers via additional Roles and States. Because the initial Butler implementation is focused on bioinformatics workflows there already exist predefined states for some common bioinformatics tools. An example of such a Role and States can be seen in the Top File mapping below:

```
'G@roles:germline':
  - biotools.freebayes
  - biotools.htslib
  - biotools.samtools
  - biotools.delly
```

Customizing Butler Configuration

When Butler is used in different environments, configurations need to change, because of differences in OS, network, and underlying analyses. In order to accomplish this, the users will typically need to create their own source code repository that will coexist with the base Butler repository. Inside that repository will be custom definitions or workflows, analyses, as well as configurations. Where it is possible to configure the system entirely via Pillar, the user should define these custom Pillar settings in their repository, but when customizations to the States are required, the user should copy the State definition from the base Butler repository into their own and customize as necessary. They should then make sure that the customized states are available to Saltstack by downloading them to the Salt Master VM.

When it comes to installing new scientific algorithms for the purposes of running workflows, the users should define any new States and Roles as necessary, and then assign them to the Worker VMs prior to calling `highstate` to ensure the software get installed properly.

3.6 Workflow System

3.6.1 Workflow System Overview

Running scientific workflows at scale is the reason for Butler’s existence. Thus, the Workflow Engine lies at the heart of the entire system. To fulfill the requirements specified in Section 3.1.5 we have selected the Airflow Workflow Engine developed by Airbnb.

Airflow Architecture

The architecture of Airbnb Airflow lends itself well to large-scale distributed processing of tasks, due to the loosely coupled nature of the system. The key component at the heart of Airflow is the Airflow Scheduler. The `airflow-scheduler` is a service that runs perpetually on a VM and examines the state of all running workflows. All workflow tasks that meet the preconditions for being runnable are immediately “scheduled” for execution. In the context of Airflow scheduling means depositing the task into a queue (running on a separate Queue Server VM) from which a Worker VM can eventually pick it up. The Worker VMs run an `airflow-worker` service which periodically polls the task queue for available tasks and when the task is runnable by a particular Worker, that Worker consumes the task message from the queue and assumes execution. In order to keep track of the status of Workers and workflow execution each Worker periodically sends heartbeat messages to the Scheduler to communicate state. The state is persisted by the Scheduler to a PostgreSQL database which runs on a DB Server VM.

Additional state information related to tracking scientific analyses is written to a separate PostgreSQL database which runs on the same DB Server.

The user can communicate with and commandeer Airflow via the Airflow CLI, as well as a Web UI. The Web UI is provided via the airflow-flower, and airflow-webserver services which can run on the same VM as the Scheduler or on a separate VM, depending on system load.

3.6.2 Workflow Definition

Requirements for workflow definitions are specified in Section 3.1.4 of this document. Conceptually, an Airflow workflow is a Directed Acyclic Graph whose vertices represent tasks and edges indicate task sequence. In its implementation an Airflow workflow is a Python program that can use any Python language construct or library. This allows the users to create workflows of arbitrary complexity and functionality.

When authoring workflows the user needs to create an instance of the DAG class

```
class airflow.models.DAG(dag_id,
    ↵ schedule_interval=datetime.timedelta(1), start_date=None,
    ↵ end_date=None, full_filepath=None, template_searchpath=None,
    ↵ user_defined_macros=None, default_args=None, concurrency=16,
    ↵ max_active_runs=16, dagrun_timeout=None, sla_miss_callback=None,
    ↵ params=None)
```

The key parameters to the DAG constructor are:

dag_id - Unique identifier for the workflow.

schedule_interval - How often the workflow is executed.

default_args - A dictionary of arguments that is passed to tasks within this workflow.

concurrency - Maximum number of concurrent workflow tasks.

max_active_runs - Maximum number of simultaneously active workflow runs.

Once the DAG is created the user can define workflow tasks. Each task is encoded as a subclass of Operator. There are three main types of Operator in Airflow:

- Operators that represent actions that need to be taken.
- Transfer operators which represent movement of data.
- Sensor operators which poll the environment for a specified condition.

All Operators are derived from the `BaseOperator` class.

```

class airflow.models.BaseOperator(task_id, owner='airflow',
 $\hookrightarrow$  email=None, email_on_retry=True, email_on_failure=True,
 $\hookrightarrow$  retries=0, retry_delay=datetime.timedelta(0, 300),
 $\hookrightarrow$  retry_exponential_backoff=False, max_retry_delay=None,
 $\hookrightarrow$  start_date=None, end_date=None, schedule_interval=None,
 $\hookrightarrow$  depends_on_past=False, wait_for_downstream=False, dag=None,
 $\hookrightarrow$  params=None, default_args=None, adhoc=False, priority_weight=1,
 $\hookrightarrow$  queue='default', pool=None, sla=None, execution_timeout=None,
 $\hookrightarrow$  on_failure_callback=None, on_success_callback=None,
 $\hookrightarrow$  on_retry_callback=None, trigger_rule=u'all_success',
 $\hookrightarrow$  resources=None, *args, **kwargs)

```

An Operator can take many parameters, the most important ones are:

dag - Reference to the DAG this task belongs to.

task_id - Unique identifier for the task.

retries - Along with several other parameters, this controls retry behaviour in case of failures.

priority_weight - Relative importance of scheduling this task compared to other tasks.

trigger_rule - Condition under which this task triggers. One of - all_success | all_failed | one_success | one_failed. This condition evaluates the state of tasks that are upstream of this one.

A large number of Operator implementations are available that simplify the creation of arbitrary workflows. Some of these are:

BashOperator - Execute a shell script.

PythonOperator - Execute a Python callable.

EmailOperator - Send an email.

ExternalTaskSensor - Wait for a task in a different workflow to complete.

HdfsSensor - Wait for a file to appear in HDFS.

HiveOperator - Execute a Hive query.

SimpleHttpOperator - Make a call to an HTTP endpoint.

PostgresOperator - Execute a PostgreSQL query.

DockerOperator - Execute a command inside a Docker container.

SSHExecuteOperator - Execute commands on a remote host.

In practice we find that the `PythonOperator` is the most versatile as it provides the ability to call arbitrary Python code which can, in turn, accomplish any of the tasks of the other operators if necessary.

Once tasks are defined their dependencies can be established by calling `task_2.set_upstream(task_1)` or `task_1.set_downstream(task_1)`. The `set_upstream()` and `set_downstream()` methods also accept lists of tasks for bulk assignment.

When a workflow is executed each task definition is transformed into a task instance - a task that is running at some point in time. Although the entire workflow may be defined in the same source file, Airflow makes no guarantees about where each task instance will run. Once a task instance is placed into the task queue technically any worker can pick up and execute that task. On the one hand this provides a limitation because it makes it difficult for tasks to exchange information between each other (due to possible remoteness), on the other hand, this model promotes scalability as it limits dependencies between tasks and simplifies scheduling.

Because no assumptions are made about which worker will run which tasks, each worker needs to have a copy of all workflow definitions that are active in the system. Furthermore, any programs that may be invoked inside a task also need to be installed on the workers. Unfortunately, Airflow does not provide any mechanisms for declaring and checking whether the programs a workflow depends on are installed and available prior to task instance runtime. This means that most bugs and issues related to dependency installation are only discovered when an actual workflow is running and fails. Thankfully, the job of installing and configuring dependencies is made relatively easy by the Butler Configuration Management System.

3.6.3 Analysis Tracker

As described in Section 3.1.6 of the requirements, a System of Record is necessary to track the scientific analyses that are undertaken using the Butler system. To fulfill these requirements we have built an Analysis Tracker module into Butler. The goal of this module is to allow the user to define analyses, specify what workflows are part of these analyses, and track the status and execution of Analysis Runs - instances of running a particular workflow on a particular data sample within the context of an Analysis. To accomplish this we have established a Run Tracking Database on PostgreSQL to persist the data, we have built a tracker Python module that implements the management of these objects, and finally, we have built a set of standard workflow tasks that the users can insert into their workflows in order to report progress to the Analysis Tracker.

tracker Python Module

The layout of the tracker module can be seen in Figure 3.10 below:

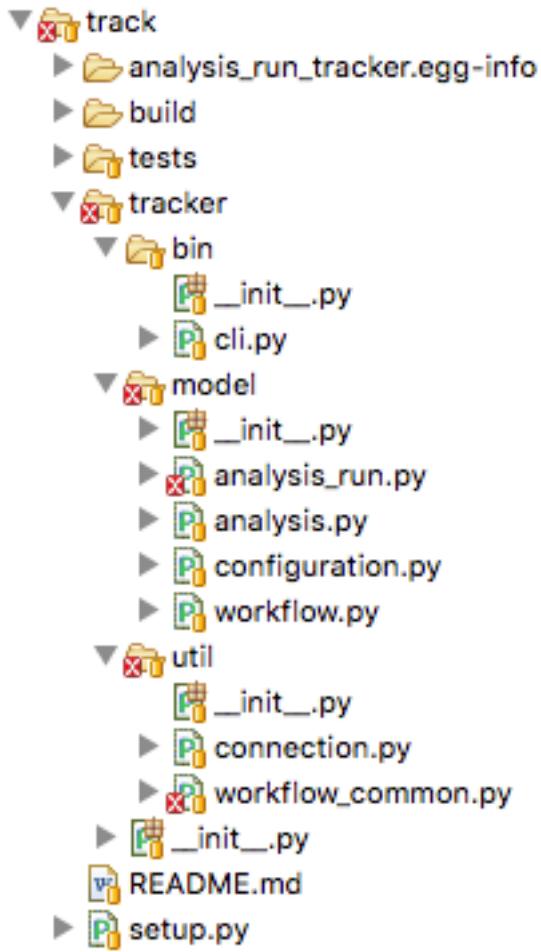


Figure 3.10: File hierarchy of the tracker module

At the root of the hierarchy are the README file and the module installation script `setup.py`. Inside the `bin` directory is the Analysis Tracker CLI implementation - `cli.py`. Inside the `model` directory lies the implementation of the main model objects - Workflow, Analysis, Analysis Run, and Configuration. We describe the first three of these objects in detail in this section and the last one in Section 3.6.4. Inside the `util` directory are utility functions - `connection.py` for connecting to the Run Tracking Database, and `workflow_common.py` for implementations of common workflow tasks.

The overall model can be seen in Figure 3.11

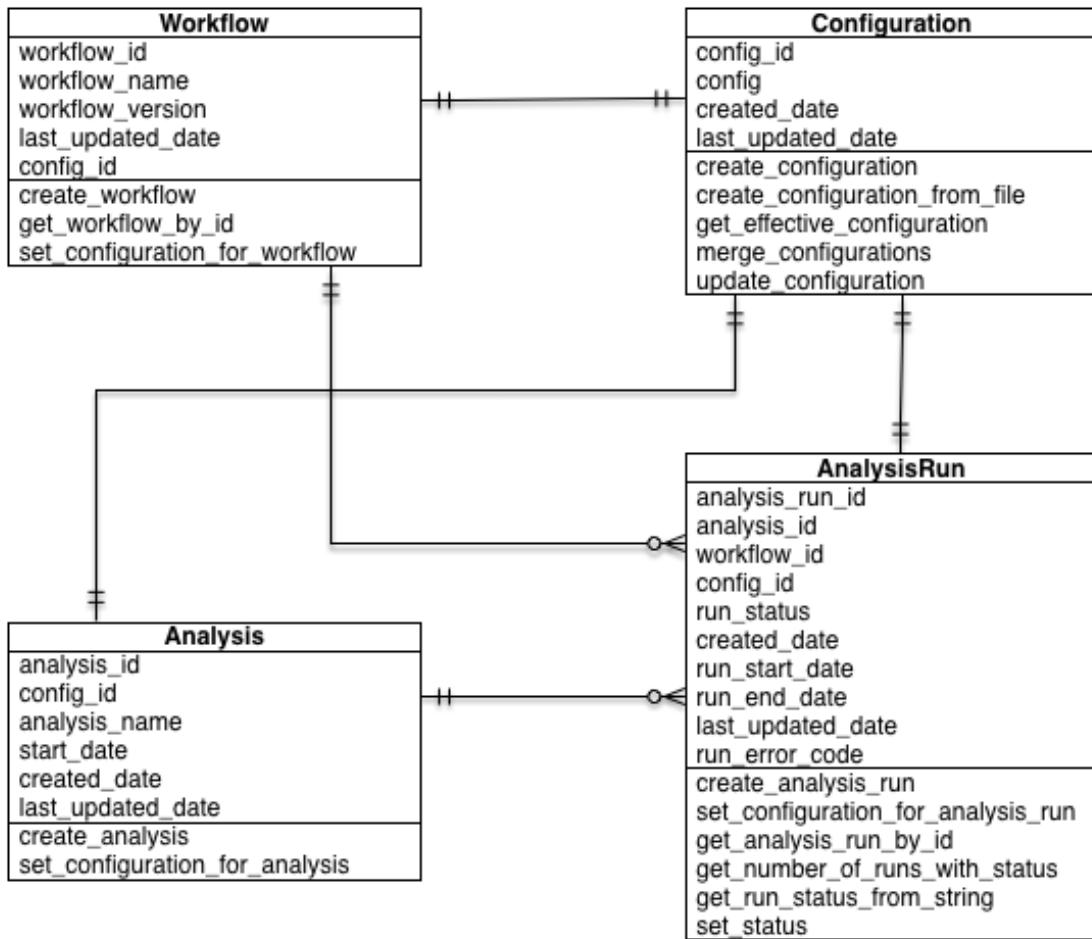


Figure 3.11: UML model of the Analysis Tracker.

The Workflow object represents a workflow definition. Every workflow managed by Butler should have a corresponding Workflow object representing it. It has the following fields:

workflow_id (UUID) - This is the unique identifier of this workflow.

workflow_name (String) - This is a human-friendly name for the workflow.

workflow_version (String) - It is important to record what version of a workflow is being used, as updates are frequently made during the workflow lifecycle.

last_updated_date (datetime) - Timestamp of last update to this object.

config_id (UUID) - The unique identifier of the corresponding Configuration object.

The Workflow object implements the following methods:

create_workflow(workflow_name, workflow_version, config_id) - Create a new workflow object with given name, version, and configuration.

get_workflow_by_id(workflow_id) - Retrieve the workflow with ID `workflow_id` from persistent storage.

set_configuration_for_workflow(workflow_id, config_id) - Update the workflow configuration to configuration with ID `config_id`.

The Analysis object represents a scientific analysis. It serves the purpose of aggregating the running of a set of workflows on a set of data samples together into a single unit of execution that can be referred to for organization purposes. The Analysis has the following fields.

analysis_id (UUID) - This is the unique identifier of this Analysis.

analysis_name (String) - This is a human-friendly name for the Analysis.

start_date (datetime) - Date of when the Analysis is meant to start.

created_date (datetime) - Date of when the Analysis was created.

last_updated_date (datetime) - Timestamp of last update to this object.

config_id (UUID) - The unique identifier of the corresponding Configuration object.

The Analysis object has the following methods:

create_analysis(analysis_name, start_date, config_id) - Create a new Analysis object with given name, `start_date`, and configuration.

set_configuration_for_analysis(analysis_id, config_id) - Update the Analysis configuration to configuration with ID `config_id`.

The AnalysisRun object represents the invocation of a particular Workflow on a particular Data Sample, within the scope of a particular Analysis. This object is central to the Analysis Tracking functionality. The AnalysisRun object has the following fields:

analysis_run_id (UUID) - This is the unique identifier of this Analysis Run.

analysis_id (UUID) - This is the unique identifier of the Analysis for this Analysis Run.

workflow_id (UUID) - This is the unique identifier of the Workflow for this Analysis Run.

run_status (int) - This integer field indicates the status of this Analysis Run. Status can be one of `RUN_STATUS_READY`, `RUN_STATUS_SCHEDULED`, `RUN_STATUS_IN_PROGRESS`, `RUN_STATUS_COMPLETED`, `RUN_STATUS_ERROR`.

created_date (datetime) - Date of when the Analysis Run was created.

run_start_date (datetime) - Date of when the Analysis Run started.

run_end_date (datetime) - Date of when the Analysis Run ended.

last_updated_date (datetime) - Timestamp of last update to this object.

run_error_code (int) - Integer pointing to an error code of Runs that ended in error.

config_id (UUID) - The unique identifier of the corresponding Configuration object.

The Analysis Run object implements the following methods:

get_run_status_from_string(run_status_string) - Translate string-based run statuses into int-based ones.

create_analysis_run(analysis_id, config_id, workflow_id) - Create an AnalysisRun and store in the database.

set_configuration_for_analysis_run(analysis_run_id, config_id) - Update the AnalysisRun configuration to configuration with ID config_id.

get_analysis_run_by_id(analysis_run_id) - Get the Analysis Run with ID analysis_run_id.

set_ready(my_run) - Set the status of a given analysis run to RUN_STATUS_READY. Only possible if the current status is not RUN_STATUS_IN_PROGRESS.

set_scheduled(my_run) - Set the status of a given analysis run to RUN_STATUS_SCHEDULED. Only possible if the current status is RUN_STATUS_READY.

set_in_progress(my_run) - Set the status of a given analysis run to RUN_STATUS_IN_PROGRESS. Only possible if the current status is RUN_STATUS_SCHEDULED.

set_completed(my_run) - Set the status of a given analysis run to RUN_STATUS_COMPLETED. Only possible if the current status is RUN_STATUS_IN_PROGRESS.

set_error(my_run) - Set the status of a given analysis run to RUN_STATUS_ERROR.

get_number_of_runs_with_status(analysis_id, run_status) - Returns the number of AnalysisRuns of a given status.

As can be seen from the description of the methods of `AnalysisRun` these objects follow a particular lifecycle that is represented by their `status` attribute. The rules that govern allowable status transitions are encoded within the series of `set_*` methods and are summarized in Figure 3.12

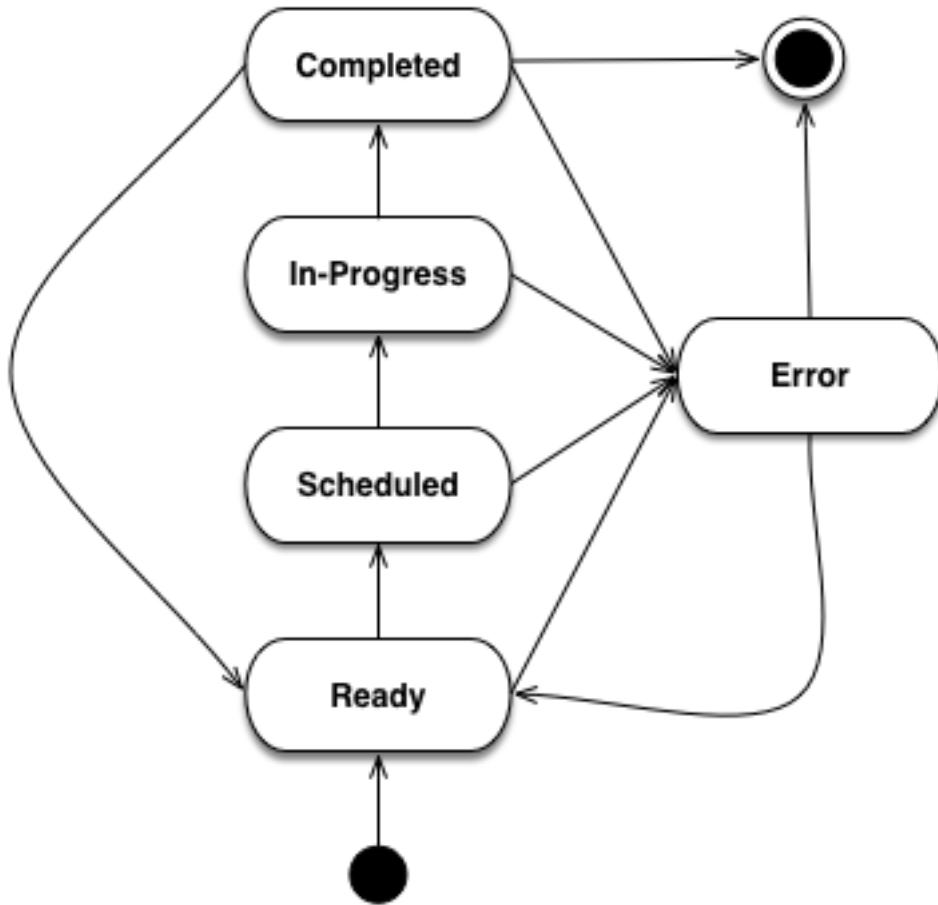


Figure 3.12: State Diagram of Analysis Run Status Transitions.

When an `AnalysisRun` is first created it does not have a status. Once the object is fully initialized it is given a Ready status, indicating that it is ready to be scheduled for execution. Once the Scheduler has scheduled the Run for execution it is given a Scheduled status. When workflow execution starts the Run is marked In-Progress. Once the Run is successfully completed it enters a Completed status. If, at any point, the Run encounters an error condition it cannot recover from, the Run Status is set to Error. When the error condition is addressed the Run status should be set to Ready so that it can start from the beginning.

The `workflow_common.py` file within the tracker module contains a number of convenience functions that workflows can use to perform common tasks. Currently the following functions are supported:

`get_config(kwargs)` - Get the Configuration supplied to this Workflow.

`get_sample(kwargs)` - Get the sample assigned to this Workflow.

`start_analysis_run(**kwargs)` - Mark the Analysis Run of this Workflow In-Progress.

`complete_analysis_run(**kwargs)` - Mark the Analysis Run of this Workflow Complete.

`set_error_analysis_run(**kwargs)` - Mark the Analysis Run of this Workflow as Error.

`validate_sample(**kwargs)` - Test whether the sample files are accessible to the workflow.

`call_command(command, command_name, cwd=None)` - Wrapper around Python's `subprocess.call` method that captures logging information.

`compress_sample(result_filename, config)` - Compress the sample with gzip.

`uncompress_gzip_sample(result_filename, config)` - Uncompress the sample.

Every workflow should begin by starting the corresponding Analysis Run, and finish by completing it to ensure appropriate tracking of information throughout the system.

The `connection.py` file is also a key component of the tracker module because it provides a means to communicate with the Run Tracking Database.

The Run Tracking Database is a PostgreSQL database this is set up to persist all of the Analysis related information into permanent storage in order to fulfill the System of Record requirements of Section 3.1.6. Specifically, the Run Tracking Database contains a relational model that corresponds to the Python objects described above. These tables are as follows:

Column Name	Type	Constraint
workflow_id	serial	PRIMARY KEY
config_id	uuid	REFERENCES configuration(config_id)
workflow_name	varchar(255)	
workflow_version	varchar(255)	
created_date	timestamp	
last_updated_date	timestamp	

Table 3.3: Database table workflow

Column Name	Type	Constraint
analysis_id	serial	PRIMARY KEY
config_id	uuid	REFERENCES configuration(config_id)
analysis_name	varchar(255)	
start_date	timestamp	
created_date	timestamp	
last_updated_date	timestamp	

Table 3.4: Database table analysis

Column Name	Type	Constraint
analysis_run_id	serial	PRIMARY KEY
analysis_id	serial	REFERENCES analysis(analysis_id)
workflow_id	serial	REFERENCES workflow(workflow_id)
config_id	uuid	REFERENCES configuration(config_id)
run_status	integer	NOT NULL
created_date	timestamp	
run_start_date	timestamp	
run_end_date	timestamp	
last_updated_date	timestamp	
run_error_code	integer	

Table 3.5: Database table analysis_run

In order to maintain a mapping between the Python objects in the tracker module and the tables in the Run Tracking Database as well as to avoid getting tied to a particular SQL dialect we utilize the SQL Alchemy Object Relational Framework. This framework allows us to avoid an explicit mapping between table columns, and object fields. Instead, SQL Alchemy is able to introspect the relational schema and add the object fields as necessary that correspond to the columns. Furthermore, updates to the Python objects are automatically translated to SQL **UPDATE** statements and executed as necessary. This framework allows us to change to a different SQL Engine if necessary, without having to change most of the code.

3.6.4 Workflow Configuration

In order to fulfill the workflow configuration and parametrization requirements described in the Analysis Configuration sub-section of Section 3.1.4 Butler implements a tri-level configuration mechanism, allowing the user to specify configurations at Workflow, Analysis, and Analysis Run levels. At runtime all three configuration levels are merged into one *effective* configuration that applies within the execution context.

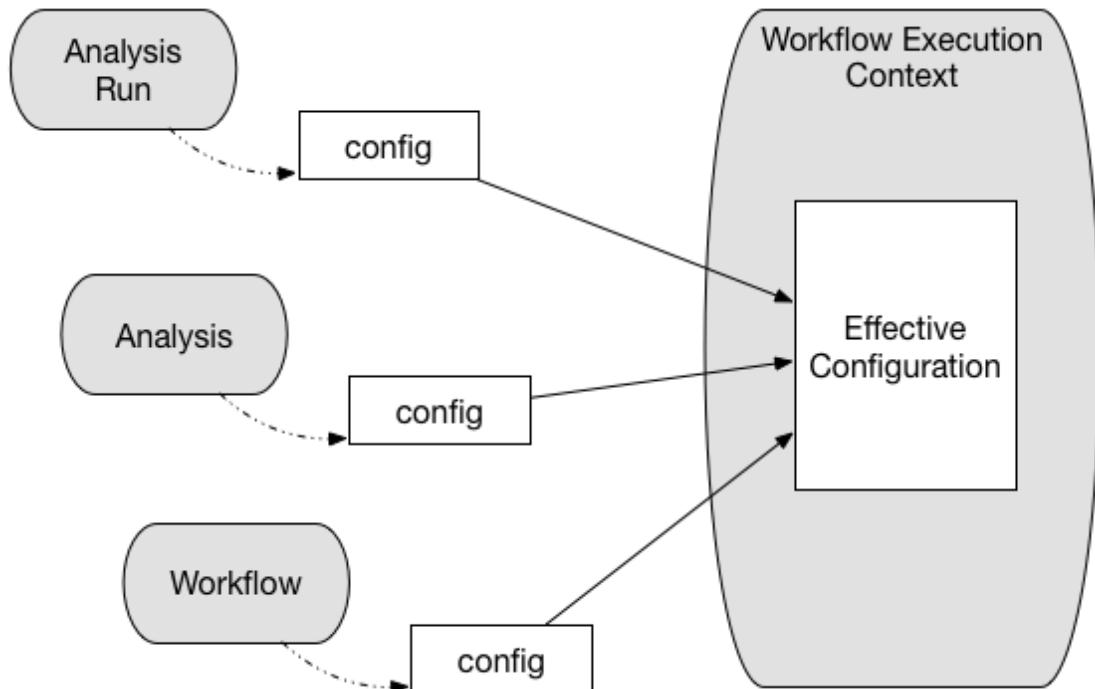


Figure 3.13: Tri-level configuration combines into an effective configuration at runtime.

The configuration facility is built into the `tracker` module, and is backed by a Run Tracking Database configuration table for persistence.

Because it is important for configuration to be both human-readable and machine-readable Butler uses the JSON format to encode configuration information. PostgreSQL, in turn, has native support for storage and deep querying of JSON values, thus making it an ideal choice for configuration persistence.

Configuration Mechanism

The mechanism by which configuration works is as follows:

A workflow author provides, together with their workflow definition, a JSON file that contains the most coarse-grained configurations related to the workflow. A system operator may customize some of these values before adding the workflow to a deployed version of Butler. Once the workflow is added to the system its accompanying configuration is persisted to the database.

A user who is running an analysis defines an analysis-level JSON file with more fine-grained configuration values. Algorithm parameters and flags are typical such values that vary from one analysis to the next. These are also persisted to the database along with the analysis definition.

As the analysis run corresponds to running a particular workflow in the context of a particular analysis on a particular sample, the user needs to generate a separate JSON configuration file for each sample in the analysis. These files will contain the most fine-grained configurations. Typical values at this level indicate where to locate the particular sample file for this run, and where to store the run results. An effective way to generate many JSON files for these runs is via a script.

When a workflow is executed on a particular sample, the JSON files corresponding

to all three levels of configuration are retrieved from the database, merged, and parsed into a Python dictionary. This dictionary is then made available within the workflow execution context to guide workflow decision logic.

3.6.5 Workflow Runtime Management

Workflow Runtime Management encompasses the tools that are available to the user for the purpose of managing workflow execution. In Butler there are three separate mechanisms that exist for this purpose. These are:

- Butler CLI
- Airflow CLI
- Airflow Web UI

Butler CLI

The Butler CLI allows the user to create the various analysis management objects described in Section 3.6.3 via a Command Line Interface. The following commands are supported:

create-workflow - creates a new workflow and supports the following parameters:

-n, --workflow_name - The name of the workflow.

-v, --workflow_version - The version of the workflow.

-c, --config_file_location - Path to the workflow configuration JSON file.

create-analysis - creates a new analysis and supports the following parameters:

-n, --analysis_name - The name of the analysis.

-d, --analysis_start_date - The starting date of the analysis.

-c, --config_file_location - Path to the analysis configuration JSON file.

launch-workflow - launches workflow execution and supports the following parameters:

-w, --workflow_id - ID of the workflow to launch.

-a, --analysis_id - ID of the analysis to launch the workflow under.

-c, --config_location - Path to a directory that contains analysis run configuration JSON files that will be launched.

update-config - Update the configuration for a workflow or analysis.

-w, --workflow_id - ID of the workflow to update.

-a, --analysis_id - ID of the analysis to update.

-c, --config_file_location - Path to the new config file.

get-run-count - Print to stdout the number of analysis runs for a particular analysis.

-a, --analysis_id - ID of the analysis to look up.

-s, --run_status - Restrict the lookup to runs with a particular status

Airflow CLI

The Airflow CLI is part of the generic Airflow framework and provides a number of commands for workflow management. We describe several of the most useful ones below:

airflow - The main Airflow CLI command, with these supported sub-commands:

webserver - Start an instance of the Airflow Web UI.

scheduler - Start an instance of the Airflow Scheduler.

worker - Start an instance of the Airflow Worker.

flower - Start an instance of the Airflow Flower, which is a monitoring tool.

clear - Clear the state of tasks that have failed or are stuck to allow them to be scheduled again.

task_state - Print out the state of a task.

initdb - Initialize an empty Airflow database.

resetdb - Reset an Airflow database to the empty state.

list_dags - List all of the available workflows.

list_tasks - List all of the tasks for a particular workflow.

Butler provides wrappers for the `webserver`, `scheduler`, `worker`, and `flower` commands so that they can be run as system services.

Airflow Web UI

The Airflow Web UI provides an interactive dashboard that allows the user to monitor the progress of workflows and workflow tasks, as well as taking some remedial actions when tasks run into trouble.

The main page of the Web UI features a listing of the available workflows, along with the breakdown of workflow tasks by status. In Figure 3.14 we see two workflows - freebayes and delly. The delly workflow has 11374 completed tasks, 865 in-progress tasks, 1 failed task, and 1 task with a failed ancestor. The user can click on any of the task statuses to navigate to a task listing page that gives a comprehensive list of tasks along with their key attributes (see Figure 3.15).



Figure 3.14: The main page of the Airflow Web UI

Task Instances															
List (61688)		Add Filter ▾		With selected ▾		Search									
×	Dag Id	equals	delly		State	equals	success		Reset Filters						
			Dag Id	Task Id	Execution Date	Operator	Start Date	End Date	Duration ▲	Job Id	Hostname	Username	Priority Weight	Queue	
□	State	Dag Id	Task Id	Execution Date	Operator	Start Date	End Date	Duration ▲	Job Id	Hostname	Username	Priority Weight	Queue		
□		delly	delly_genotype	07-18T14:26:48.824685	PythonOperator	07-20T11:46:11.877037	07-22T23:03:55.974008	2 days, 11:17:44.096971	1092483	worker-80	airflow	2	default		
□		delly	delly_genotype	07-20T12:05:30.057098	PythonOperator	07-20T12:28:21.295523	07-22T23:00:34.527860	2 days, 10:32:13.232337	1098775	worker-30	airflow	2	default		
□		delly	delly_genotype	07-20T12:08:11.048047	PythonOperator	07-20T12:28:20.429414	07-22T22:32:56.574494	2 days, 10:04:36.145080	1098587	worker-33	airflow	2	default		
□		delly	delly_genotype	07-19T08:03:04.375128	PythonOperator	07-20T11:46:12.363633	07-22T19:34:11.701710	2 days, 7:47:59.338077	1092493	worker-21	airflow	2	default		
□		delly	delly_genotype	07-20T12:07:51.023870	PythonOperator	07-20T12:28:19.210927	07-22T18:16:24.588959	2 days, 5:48:05.378032	1098617	worker-34	airflow	2	default		
□		delly	delly_genotype	07-20T12:07:50.947435	PythonOperator	07-20T12:28:25.091254	07-22T17:40:14.522576	2 days, 5:11:49.431322	1099041	worker-29	airflow	2	default		
□		delly	delly_genotype	09-07T12:55:21.420778	PythonOperator	09-07T13:15:39.572854	09-09T17:27:20.653645	2 days, 4:11:41.080791	1467149	worker-113	airflow	2	default		

Figure 3.15: Listing of task instances for the freebayes workflow with status of "success".

Clicking on one of the task instances will bring up a graphical view of the workflow the task belongs to and allow the user to further investigate that workflow's execution (see Figure 3.16). This view shows the status of all tasks in the corresponding workflow instance as well as providing links to various reports.

When the user clicks on a particular task instance within the workflow execution a popup window allows them to take a number of actions, such as forcing the task to be run immediately, clearing the task state (for failed tasks) so that it can be run again, or marking the task as successfully completed (possibly with all upstream and downstream tasks). These options are depicted in Figure 3.17

3.7 Operational Management

Managing large fleets of Virtual Machines as they perform data analysis at scale across multiple cloud computing environments is a major challenge due to the sheer number of possible scenarios that could lead to the system crashing, stalling, or otherwise falling out of control, with the negative impact on the end user in terms of project costs and timeline increasing with the scale of the computation being undertaken. The tools available to the user to detect and deal with these issues thus form a key component of a comprehensive analysis framework and set Butler apart

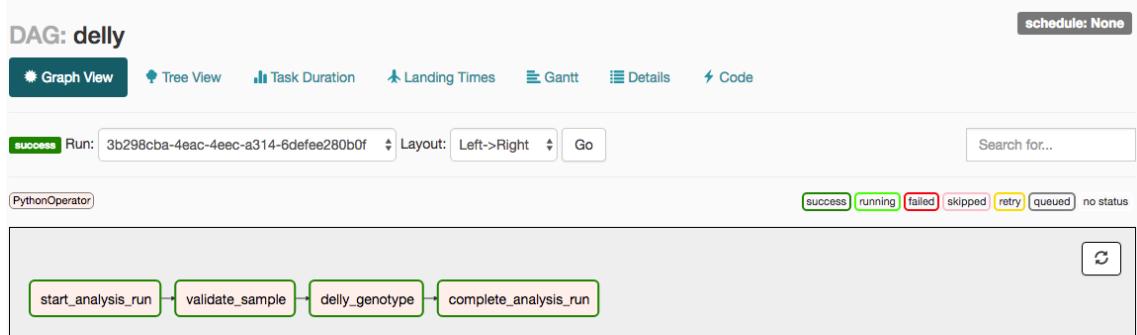


Figure 3.16: Details of the execution of a delly workflow with ID 3b298cba-4eac-4eec-a314-6defee280b0f.

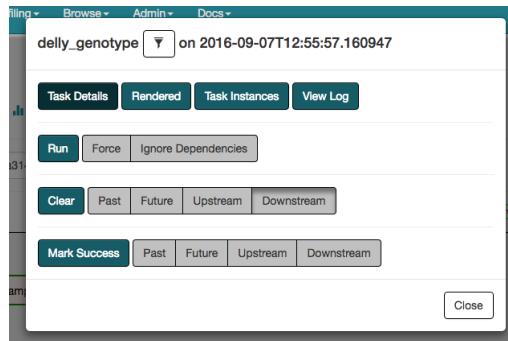


Figure 3.17: A list of actions that can be taken on a workflow task via the Web UI.

from other frameworks in this space.

In general, the Operational Management tools fall into three categories, those that collect observations about the state of each component in the system at runtime, those that aggregate this data and present it to the user in the form of queryable databases and management reports, and those that take automated remedial steps to resolve identified issues. Furthermore, as specified in the requirements of Section 3.1.7 we delineate two major sources of data that is indicative of system state - System Metrics, and Server Logs. While metrics provide more of a coarse-grained view of the overall health of a particular Virtual Machine, server logs can give much more of a fine-grained view of the underlying system at an application level, and down to individual lines of code that are running at any given time. Butler has dedicated components for the collection and management of these data sets and we describe these next.

3.7.1 Monitoring Metric Collection

The overall health of any VM in a cloud computing cluster can be evaluated and ascertained with respect to a set of key metrics that are observable at runtime. Some metrics are general enough that they apply to all Virtual Machines, these include measurements of CPU utilization, RAM, Disk, and Network usage. Other metrics are more specific to the role that the VM is playing within the cluster. A Database Server will benefit from having the number of open DB connections, transaction rate, rollback rate, and average query runtime monitored. Other entities such as Web Servers and Queues have their own unique metrics of interest.

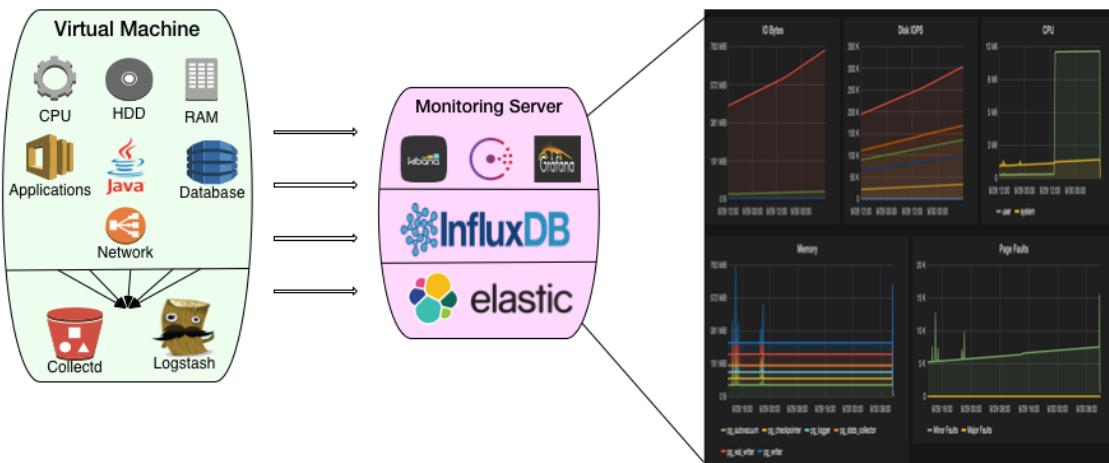


Figure 3.18: Metric monitoring system components

Each VM runs a metric collection daemon called `telegraf`[3] which is an Open Source package that is able to make periodic measurements of a large number of system metrics and ship them off to a centralized Metrics Store. The definition for which metrics are collected is specified in a special configuration file. An example of such configuration is demonstrated in Listing 19. This listing provides examples of the setup for generic metrics like CPU, and memory, as well as for more special metrics, like those collected for PostgreSQL databases.

Because we are interested in observing not only the metrics as they are measured in the present, but also the dynamics of how metric values change over time, we need a mechanism for persisting the metrics values. For this purpose the Monitoring Server component of Butler ships with an instance of a database product called InfluxDB[98] which is an Open Source database system that is optimized for recording time series data. The configuration for the `influxdb` output plugin in Listing 19 demonstrates how InfluxDB server URL is provided to Telegraf to enable metrics persistence in this centralized data store.

InfluxDB is a scalable database system that can operate in a distributed manner as a Raft[165] cluster. Data is written into shards based on a data retention policy. The underlying engine stores the data as a Time-Structured Merge Tree (TSM) which is a customization of the Log-Structured Merge-Tree[162] data structure. To the end user the data is organized as a collection of **points**. Each point has a **timestamp**, belongs to a **measurement**, and records a set of **tag_values** that correspond to a set of **tag_keys**. A particular **measurement** coupled with a retention policy then forms a **series**.

As an example, when tracking RAM, our **measurement** is called `memory_value`. The set of **tag_keys** is:

- host
- type
- type_instance

An example set of **tag_values** might be:

- worker-145

- memory
- free

The actual **points** corresponding to this combination of **tag_values** and collected over the period of 1 minute might look as follows:

Query:	SELECT * FROM "memory_value" WHERE "type_instance" = 'free' and time > now() - 1m and host = 'worker-145';
	Query Templates ▾

memory_value				
time	host	type	type_instance	value
2016-10-12T12:39:59.207858Z	"worker-145"	"memory"	"free"	361644032
2016-10-12T12:40:09.207854Z	"worker-145"	"memory"	"free"	361897984
2016-10-12T12:40:19.207859Z	"worker-145"	"memory"	"free"	361644032
2016-10-12T12:40:29.20786Z	"worker-145"	"memory"	"free"	361517056
2016-10-12T12:40:39.207859Z	"worker-145"	"memory"	"free"	361627648
2016-10-12T12:40:49.20788Z	"worker-145"	"memory"	"free"	361627648

Figure 3.19: InfluxDB query results showing free memory on host worker-145 collected in a 1 minute time window.

As can be seen in Figure 3.19, data is queried via an SQL-like dialect which is accessible via a Web UI or via a REST[70, Chapter 5] API.

3.7.2 Monitoring Visualization

The metrics collection system is collecting 50 different metrics per host on average, sampled at intervals of 10 seconds. Given a cluster of 200 Virtual Machines the monitoring system collects and stores 86,400,000 data points in a 24 hour time period. This volume of data is quite difficult for the user to comprehend and make use of, and Butler provides visualization tools to enable the display of aggregate statistics based on the monitoring data using a Graphical User Interface. The main goal of the visualizations is to give the user an overview of the trends observed within the compute cluster with respect to a set of representative performance metrics, and to alert the user to any conditions that threaten the health of Virtual Machines and the scientific analyses they run.

Visualization of performance metrics data is accomplished in Butler using an Open Source framework called Grafana[163]. This framework provides a Web Interface that is able to connect to an instance of an InfluxDB database, issue queries and render the query results as dashboard of various graph styles, including line graphs, bar graphs, pie charts, and others. Reports generated by Grafana support parametrization of values as well as drill-through.

Butler ships with a number of pre-built reports and supports the addition of custom reports if necessary.

In general, Grafana is a website that the user logs onto to either author or view reports. When authoring reports the user needs to define a Data Source to connect the reports with data. In the case of Butler, the Data Source is connecting to the InfluxDB metrics database using the InfluxDB REST API.

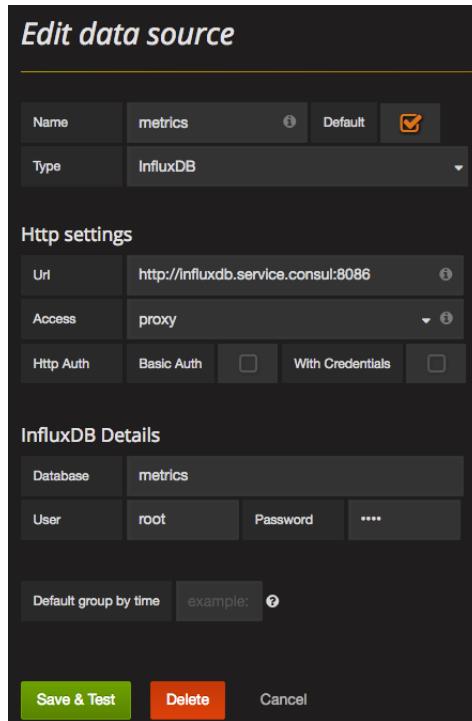


Figure 3.20: InfluxDB Data Source creation for Grafana.

Once a connection to the Data Source is established the user can begin building dashboards. Dashboards are generally laid out in a grid-like fashion as a series of panels that are arranged into rows and columns. Each panel contains a graph. Data is fed into the graph by means of an InfluxDB Query Language query, possibly with additional parameters specified by the report. In addition to the query itself, the user can customize the axes, legend, and other attributes of each graph. Once the report is ready it can be exported into a JSON format and checked into a source code repository so that it can persist if the VM hosting the Grafana instance needs to be recreated.

Butler comes with a set of basic dashboards defined out-of-the-box that will aid the user in monitoring of overall system health. The most important dashboards are:

Cluster Overview - Gives a high level overview of the health of the entire cluster, tracking metrics such as load (blended health metric), CPU utilization, Memory, Disk IOPS, Network Packet Rate, Disk Read/Write Rates, and Disk Space

Salt Master - Detailed monitoring of the Salt Master VM, including CPU, Memory, Network Packet Rate, Disk Read/Write Rates, Disk IOPS, and Disk Space.

Database Server - Detailed monitoring of the Database Server VM, including DB-specific metrics such as - DB Connections, Number of Transactions, Number of Queries, Number of Query Plans, Number of Rows, DB Size on Disk, as well as the general VM health metrics.

In practice the Cluster Overview dashboard is the most consistently useful report because it provides an at-a-glance view of the health of the entire cluster. Figure



Figure 3.21: Editing a Grafana dashboard panel.

3.22 demonstrates a typical scenario of cluster usage during normal operation. It is evident that the cluster starts off without any appreciable load, once a set of workflows is scheduled system load increases across the entire cluster and remain stable until, towards the end of the execution, machines start running out of work, and load of the system gradually decreases. Throughout the execution the memory profile is relatively stable and consistent across the machines in the cluster. Although the CPU profile appears spiky, this is the natural CPU utilization profile for the particular set of workflows that were executing at the time the image was captured. Figure 3.23, on the other hand, demonstrates an example of how a cluster-wide issue can be identified via the Cluster Overview dashboard. Here we see a similar profile at the beginning of workflow execution, where the jobs get kicked off and load is stable, but at around 00:30 we start seeing uncharacteristic spikes in system load, with sporadic doubling of load metric values. Furthermore, the CPU utilization appears to drop to 0 at the same time, and the memory profile is highly irregular. This pattern signifies to the user that an issue is affecting the cluster during this particular time, and further investigation is needed. With that knowledge, the user can use other reports to try to pinpoint the source of the issue, or it may be necessary to directly log on to the affected VM hosts to investigate further.

Because collectd allows collecting metrics from a particular named process, and because Grafana allows the creation of new dashboards on the fly, it is common and convenient for the user to develop a set of custom dashboards that are targeted towards the specific workflows that they are running, thus providing much more fine-grained visibility into the runtime behaviour of the system. Together with the built in reports these custom dashboards provide a powerful and flexible set of capabilities



Figure 3.22: Cluster Overview dashboard during normal operation.

for successful operational management of the Butler system.

3.7.3 Server Log Collection and Visualization

Almost every application that runs on a computer is generating some sort of log file. On Unix-based environments most system-level applications will write to a common log known as syslog. But many other applications will write their own custom logs to their own specific locations. Messages written to a log file typically run the gamut from INFO statements that mark the normal operation of an application, all the way to ERROR which signify error conditions. Thus, a log file, potentially provides a wealth of information about both normal operation and system issues as they occur, and is typically one of the most reliable sources for information on application crashes. On the other hand, when operating a complex distributed system, such as a large scale workflow execution framework which runs on hundreds of Virtual Machines, the number and size of logs can become overwhelming to the point of ceasing to remain useful.

Because of the potentially extremely high value of the information contained in server logs, we deploy a system of log harvesting and centralized storage that enables the Virtual Machines that are part of Butler to parse the logs that are being generated locally for interesting events, and send those events to a centralized search index which is amenable to efficient querying and visualization. Although the three tools that we use to solve the centralized logging problem have been developed independently, they have since been acquired by a single company Elasticsearch BV. These three tools form what is known as the ELK stack - Elasticsearch[19], Logstash[201], and Kibana[106].

Each Virtual Machine in the cluster runs a log shipper - Filebeat. It is responsible for finding, harvesting, and locally aggregating logs.



Figure 3.23: Cluster Overview dashboard demonstrating a cluster-wide issue.

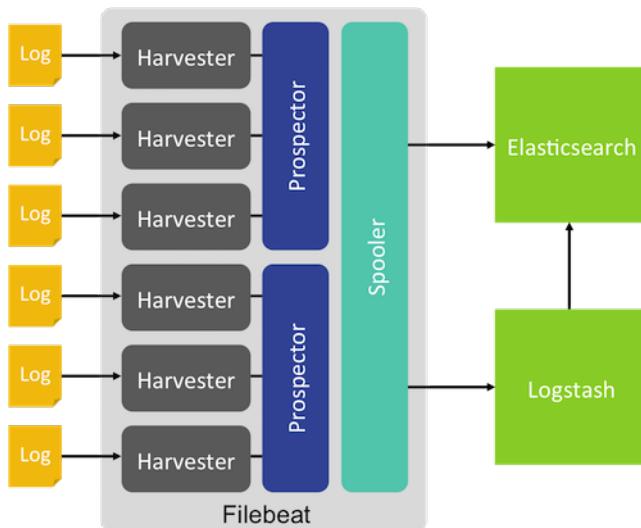


Figure 3.24: Filebeat Architecture (taken from <https://www.elastic.co>).

As shown in Figure 3.24, Filebeat consists of a set of Prospectors (see Listing 20) which monitor and search log directories specified in the Filebeat configuration file. Corresponding to each log file that is found by Prospectors a separate Harvester is started which is responsible for ingesting the log file and sending the information to a Spooler. The Spooler aggregates information sent from Harvesters and forwards it onto Logstash for further parsing.

Logstash runs on a separate centralized server and is responsible for parsing the logs forwarded from Filebeat and sending the parsed information on to the Elasticsearch index. The parsing is accomplished via 3 plugins - Input Plugin, Filter Plugin, and Output Plugin (see Figure 3.25). All three plugins are configured via the logstash configuration file.

The Input Plugin specifies where to listen to input data from. In the case of Butler we are expecting data to arrive in Filebeat format on port 5044.

The Filter plugin specifies how to parse the log file i.e. which messages from each type of log file we are interested in. The interesting messages are identified using a series of regular expressions known as grok patterns.

The output plugin then specifies how to pass the filtered messages on to the Elas-

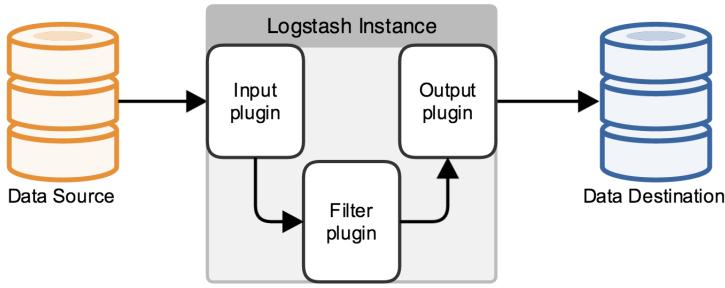


Figure 3.25: A Logstash processing pipeline (taken from <https://www.elastic.co>).

ticsearch engine for indexing.

Elasticsearch is a general purpose scalable text indexing and search engine that supports clustering and sharding of data. Given its longterm use as a storage engine for log data and its scalability it is a great fit for Butler's log storage needs. Elasticsearch works by storing JSON formatted documents (in this case log messages) into an searchable index.

Just as it is difficult to grasp and analyze performance metrics due to the number of data-points generated, it is as difficult to grasp log messages from a large cluster. We utilize a similar set of visualization tools to the ones we used for metrics, to solve this problem for server logs within Butler. The Kibana dashboarding framework allows us to create graphical dashboards that visualize log events of interest, as well as providing a web-based query interface to the Elasticsearch log messages index. Figure 3.26 shows a dashboard used in Butler for monitoring the Database Server.

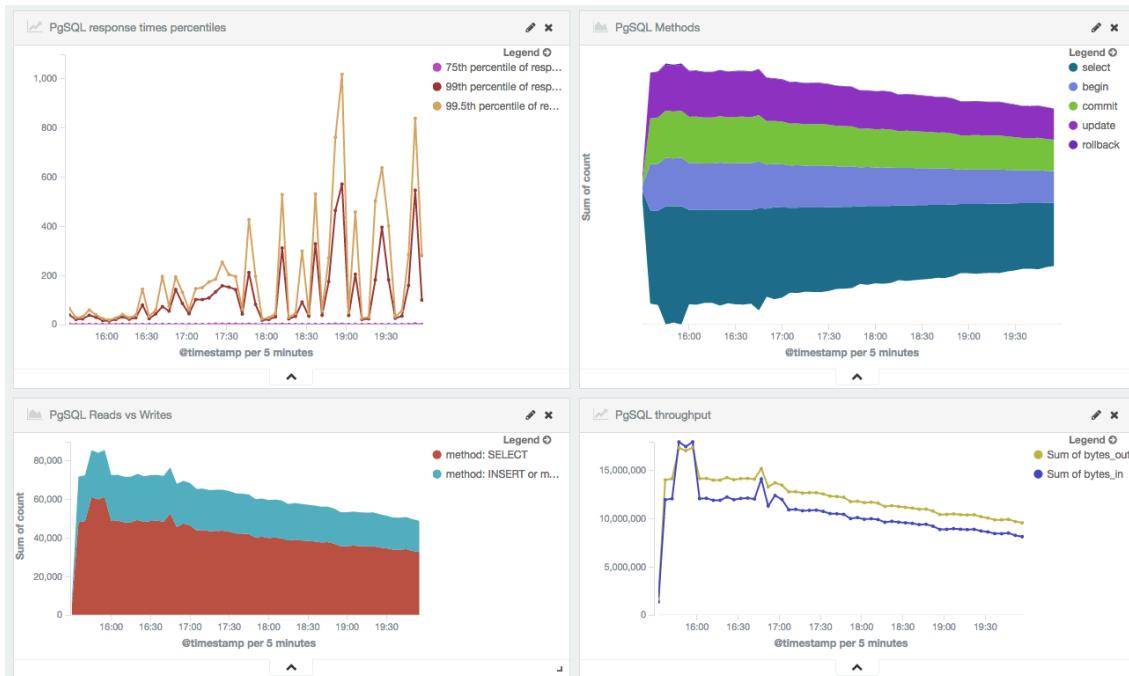


Figure 3.26: Kibana dashboard for PostgreSQL monitoring.

3.7.4 Self-Healing

The self-healing system within Butler builds on top of the metrics collection system to deliver features specified in Section 3.1.7 of the requirements. Specifically, it aims to provide a comprehensive set of tools for detecting operational anomalies at multiple levels of the system and taking automated remedial steps to fix these anomalies in order to minimize their effects. Figure 3.27 demonstrates the components of this system.

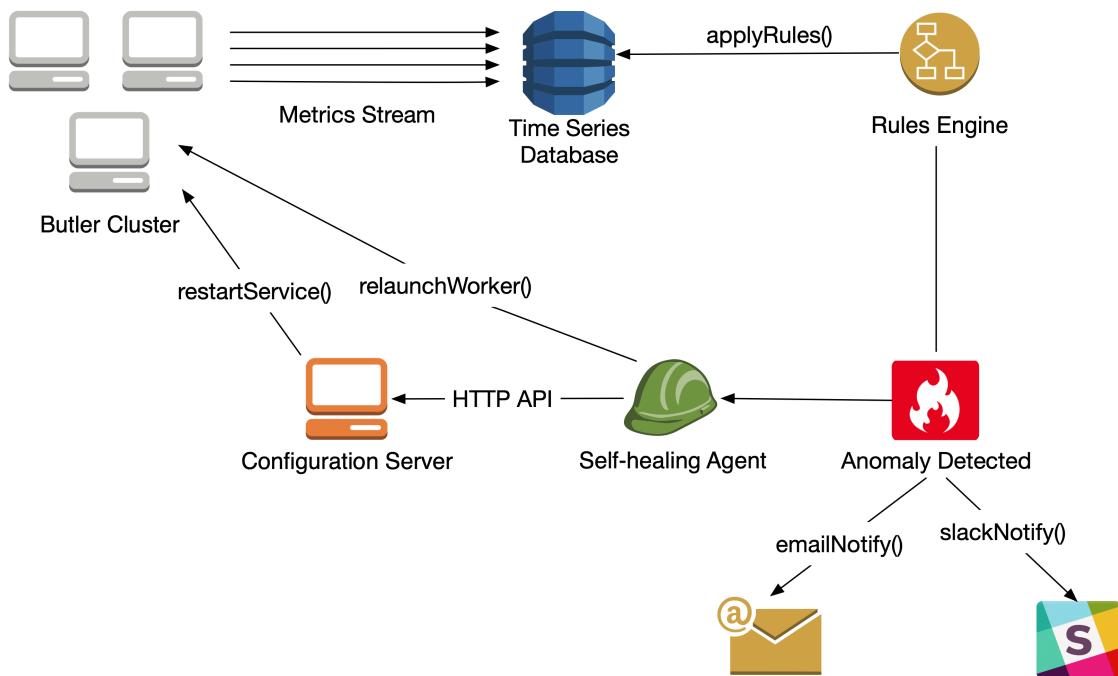


Figure 3.27: The Butler self-healing system.

Capturing Metrics

The input to the self-healing system is the set of metrics that is collected from all hosts into InfluxDB via telegraf. In order to facilitate the detection of anomalies at infrastructure, service, and payload levels it is necessary to collect several types of metrics.

Infrastructure-level issues are a wide class off issues that can represent the failure or entering into a bad state of individual components of the VM or the entire VM. Detecting these relies on capturing of a variety of basic metrics such as CPU, memory, disk, network, and others. Failure of the entire VM can be detected by the complete absence of a metrics signal from the affected machine. In practice, Butler uses the lack of updates to the `uptime` metrics as a signal that the machine has gone offline. Tracking of individual services is facilitated by monitoring their respective processes. There are several possibilities for doing this offered by telegraf with its `procstat` input plugin and these are used to capture the necessary processes in Butler. See listing 2 for the variety of ways to capture a process employed within Butler. Here, a process can be targeted by capturing the name of its executable, its PID file, a grep pattern on the process name, or the user that owns the process. Capturing the process in this way will generate a set of process-specific metrics that record

the process's CPU usage, memory, context switches, and others that can be used in detecting and diagnosing issues.

Listing 2: Butler Analysis configuration for VCF filtering.

```

1  [[inputs.procstat]]
2  exe = "grafana-server"
3
4  [[inputs.procstat]]
5  pid_file = "/var/run/salt-api.pid"
6
7  [[inputs.procstat]]
8  pattern = ".*airflow scheduler.*"
9
10 [[inputs.procstat]]
11 user = "airflow"

```

Tracking detailed metrics at the payload level beyond what is offered by procstat is, by definition, not possible to do in a completely generic fashion because every user of the framework will potentially run their own custom payloads that need to be individually instrumented with metrics. In usage scenarios where the payload is of a particularly experimental nature (such as scientific algorithms that are in early stages of development), or the analysis is performed at a large scale where the runtime is expected to be in months, and there is access to the underlying tools' source code, it may be desirable to add metrics emitting code within the individual tools that are run by the Butler framework. This is supported in Butler via the popular `statsd`[4] package which has client libraries in every major programming language and allows any software to capture and emit several kinds of custom metrics in a format that is understood by a wider variety of metrics collection back-ends, including InfluxDB. The metric types that can be captured are:

Counter - A simple counter that allows arbitrarily incrementing and decrementing a value.

Timer - A metric that measures passage of time.

Gauge - A metric that emits a measurement of an arbitrary value.

Set - A metric that counts unique occurrences of an event.

Using the basic metric types above one can construct metrics capturing schemes of arbitrary complexity that subsequently feed into the self-healing capabilities of the framework.

Defining Anomalous Operating Conditions

In order to perform self-healing the Butler framework needs to be able to detect when anomalous conditions occur. This is most easily accomplished via a rules-based system over the universe of metrics where each rule is a predicate of the form:

$$f(\vec{\mathbf{M}}_n) \text{ op } C \quad (3.1)$$

given:

$M = \{M_t; t \in T\}$ - a metric time series

$\vec{M}_n = [M_{t-n+1} \ M_{t-n+2} \ \dots \ M_t]$ - a vector of n most recent observations of M

$f: \mathbb{R}^n \mapsto \mathbb{R}$ - a function of \vec{M}_n that typically computes a summary statistic over a moving window of the most recent n data points, such as mean, min, max, etc.

$op \in \{>, <, >=, <=, ==, !=\}$ - is a comparison operator

$C \in R$ - is an arbitrary constant

Thus, each rule continuously (discretized by metric sampling frequency) computes a summary statistic over a sample of n most recent observations of metric of interest M , and compares it to a threshold value C , firing off an event when the comparison specified in op succeeds. The full collection of rules then constitutes a full specification of the set of anomalous conditions for which automated action can be taken.

We implement this rule-based system in Butler by leveraging an open-source product called Capacitor[5] which provides a rule definition and execution environment that integrates well with our metrics data store InfluxDB and provides a wide range of options for event handling, including completely custom event handlers, which we make use of to implement the automated healing functionality in Butler. Capacitor rule definitions make use of a Domain Specific Language (DSL) called TICKscript to implement the specification of rule predicates described in 3.1.

TICKscript makes use of the concept of nodes and pipelines to organize rules as Directed Acyclic Graphs (DAGs) where processing flows forwards in a single direction, and never backwards. Each node describes a particular data processing stage, and the pipeline is a chain of nodes. Each invocation of a rule represents a particular path through the graph.

In listing 21 we see an example of a TICKscript used in Butler to keep track of the CPU allocation on each host and send results to a log file. Key threshold levels are specified at the top of the script as well as the look-back period and evaluation frequency. A query for fetching appropriate data is specified and stored in the `data` variable as a stream. Here we access the `metrics` database (of our centralized InfluxDB data store), and we are interested in the `cpu_value` measurement, expressed as a percentage. The actual metric measures the idle CPU percentage, and we are interested in the used CPU percentage, so we subtract the metric value from 100, and compute the mean value over our selection window giving it an alias name of `stat` that we can refer to later.

With the data appropriately selected we can turn our attention to specifying appropriate comparisons to the thresholds that had earlier been established, and defining event triggers and handling logic. In addition to the already existing mean value we use the built-in `sigma` function to calculate the distance, in standard deviations, from the mean of the current `stat` value, giving it an alias `sigma` as we are interested in capturing and alerting not only on high absolute CPU utilization levels but also high variance within the metric of interest. Each rule can trigger at three levels of severity - info, warning, and critical and we specify a separate threshold for each level. Thus, the CPU rule is triggered at a critical level when average CPU utilization over the last 10 seconds is greater than 90%, or when the observed CPU utilization is greater than 3.5 standard deviations away from the mean CPU utilization. Other thresholds are set similarly. In this particular example we are interested

in collecting the instances when rules are triggered into a simple log file, thus we specify a path to such a log file as part of the alert definition.

A key mode of failure for the system is when a critical service fails or an entire virtual machine becomes unresponsive and unreachable. In order to capture these conditions we make use of a special type of rule within `kapacitor` called a *deadman*. This type of rule is triggered when no signal is received for a specific metric for a predefined period of time. In Listing 22 we provide an example of a TICKscript that detects the deadman condition for an entire VM. This script uses a higher level of parametrization than the CPU utilization script of Listing 21 where most of the parameters are specified as variables at the top of the script and the script structure is encoded below. The metric measurement that we base the rule on is called `system` and it tracks low-level stats such as uptime for a VM. The data selection section of the script selects values from the `system` measurement grouped by individual host that they originate from.

The rule is triggered when no data is received for the given metric via the `deadman(threshold, period)` node. In this case, when the rule is triggered there are several handlers that are registered to handle the resulting event. The first handler uses Slack API[6] to post a notification message to the `#embassyalerts` Slack channel using a predefined message template with key information about the outage, such as in Figure 3.28. The type of message changes depending on whether the rule condition has triggered or returned back to normal.

Monday, March 26th

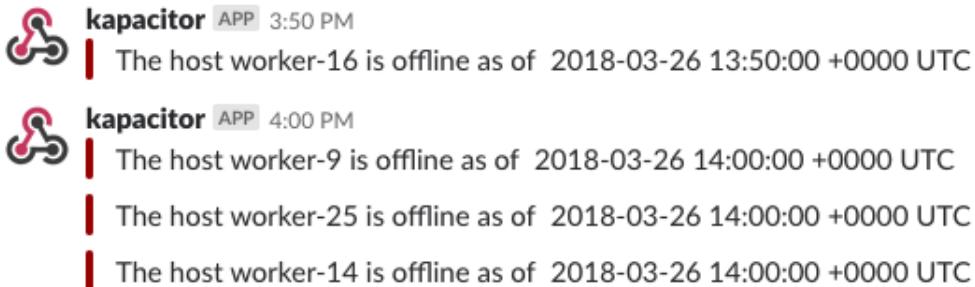


Figure 3.28: Self-healing alerts sent to a Slack channel.

The second handler invokes the actual Butler Self-Healing Agent to destroy the host that has become defunct and to launch a new VM to replace it. The details of the agent are provided in the next section. The third handler for this event actually generates a new metric that keeps track of rules that have been triggered and stores the metric in a separate measurement in InfluxDB. Since we want to have comprehensive information about when error conditions occur the `alerts` measurement gives us a system of record for these events.

Butler defines a variety of rules using TICKscript that capture anomalies that are actionable out of the box, such as those that detect when a variety of services go offline, including - PostgreSQL service, Airflow Scheduler, Airflow Webserver, Airflow Worker, RabbitMQ, Nginx, Grafana, as well as when an entire worker VM goes offline. Each rule needs to be registered with kapacitor and Butler uses the standard Saltstack configuration mechanism to register these rules within the `deploy_ticks` Salt state. There exists an open source Graphical User Interface called chronograf[7]

11 Alert Rules					+ Build Alert Rule
Name	Rule Type	Message	Alert Handlers	Task Enabled	
Airflow Scheduler Heartbeat	Deadman	Airflow Scheduler heartbeat mis...	exec, slack (default)	<input checked="" type="checkbox"/>	
Airflow Webserver Deadman	Deadman	Airflow Webserver not respondin...	exec, slack (default)	<input checked="" type="checkbox"/>	
Airflow Worker Deadman	Deadman	Airflow Worker on {{index .Tags "..."}} is not responding!	exec, slack (default)	<input checked="" type="checkbox"/>	
Chronograf Deadman	Deadman	Chronograf not responding! {{.Ti...	exec, slack (default)	<input checked="" type="checkbox"/>	
Consul Deadman	Deadman	Consul is down on {{index .Tags ...}}	exec, slack (default)	<input checked="" type="checkbox"/>	
DB Server Heartbeat	Deadman	DB Server not responding! {{.Na...	exec, slack (default)	<input checked="" type="checkbox"/>	
Grafana Deadman	Deadman	Grafana offline at {{.Time}}	exec, slack (default)	<input checked="" type="checkbox"/>	
Host Deadman	Deadman	The host {{index .Tags "host")}} is...	exec, slack (default)	<input checked="" type="checkbox"/>	
Job Queue Heartbeat	Deadman	Job Queue is not responding ({{....	exec, slack (default)	<input checked="" type="checkbox"/>	Delete
Nginx Deadman	Deadman	Nginx not responding! {{(.Name)}}	exec, slack (default)	<input checked="" type="checkbox"/>	
Salt Master Deadman	Deadman	Salt Master is not responding! {{....	slack (default)	<input checked="" type="checkbox"/>	

Figure 3.29: List of rule definitions within chronograf.

that allows for visualization and manual inspection and editing of kapacitor rules and the underlying InfluxDB metrics, and Butler makes use of this GUI, see Figures 3.29, 3.30.

1144 Alerts					<input type="text"/> Filter Alerts...
Name	Level	Time	Host	Value	
Airflow Worker Deadman	●	2018-10-23T11:10:00.000Z	worker-1	0	
Consul Deadman	●	2018-10-23T10:10:00.000Z	worker-35	0	
Consul Deadman	●	2018-10-23T10:00:000Z	worker-27	0	
Airflow Webserver Deadman	●	2018-10-28T13:35:00.000Z	tracker	9	
Airflow Webserver Deadman	●	2018-10-28T13:34:00.000Z	tracker	0	
Airflow Scheduler Heartbeat	●	2018-10-28T13:33:00.000Z	tracker	5	
Airflow Webserver Deadman	●	2018-10-28T13:33:00.000Z	tracker	9	
Airflow Webserver Deadman	●	2018-10-28T13:32:00.000Z	tracker	0	
Airflow Scheduler Heartbeat	●	2018-10-28T13:31:00.000Z	tracker	0	
Airflow Webserver Deadman	●	2018-09-30T16:37:00.000Z	tracker	13	
Airflow Scheduler Heartbeat	●	2018-09-30T16:37:00.000Z	tracker	8	
Airflow Scheduler Heartbeat	●	2018-09-30T16:35:00.000Z	tracker	0	
Airflow Webserver Deadman	●	2018-09-30T16:35:00.000Z	tracker	0	

Figure 3.30: Butler alert history.

Executing Self-Healing

Metrics are comprehensively collected within Butler and anomaly detection signals are generated via a set of rules as previously described, but the actual automated self-healing capabilities require a separate component and no generic open source components exist that accomplish this. We thus implement a custom Self-Healing Agent within Butler. The functionality is exposed as a Python package with a CLI interface that is able to accomplish a number of healing tasks on various machines in the Butler cluster.

The self-healing agent relies on the configuration management capabilities available in Butler via Saltstack in order to communicate healing instructions to other hosts. Specifically, Saltstack provides a programmatic interface through which a salt-master can be controlled. This interface is called salt-api. In addition, there

exists a module called `pepper` that allows for the salt-api to be communicated with by a remote program via openssl. Butler uses this mechanism to communicate with, and restart various failed services in an automated fashion. For instance, in the case of a failed Airflow Scheduler service (see Listing 23), the local telegraf metrics collector monitors the `airflow-scheduler` service, and sends updated metrics to the monitoring-server every 10 seconds, where the data is stored in InfluxDB. A capacitor rule runs on the monitoring-server every 10 seconds and evaluates whether the `airflow-scheduler` is still sending data. If there is no data for longer than 3 minutes, the rule is triggered. As a result the Self-Healing Agent is requested to restart the `airflow-scheduler` service. The agent itself does not know how to communicate with other machines in the cluster, but knows how to communicate with the salt-master. It establishes a salt-api connection to the salt-master and asks for the VM that has the tracker role to restart its `airflow-scheduler` service. The salt-master knows about all of the VMs in the Butler cluster and issues the restart service command, placing the service back in a functioning state. A Slack notification is sent to alert a human operator that an outage has occurred and has been resolved. The total time of the outage is slightly longer than 3 minutes.



Figure 3.31: Butler timeline of alerts for high CPU utilization.

Although, as described above, there are quite a few steps involved between the initial stoppage of a running service and a successful automated restart, the process is relatively straightforward. Restarting a failed VM, on the other hand, requires quite a bit more effort and careful management. Detecting when an entire VM fails is not really possible in a centralized uniform manner, since a failed VM is completely unresponsive by definition, yet other conditions such as network outages and failures of the metrics collection system may appear in an identical manner to an outside observer. Thus, VM failure detection is prone to false positive signals, and because re-creating VMs from scratch is a costly and time consuming process it is important to use this feature sparingly.

For every VM a set of basic metrics is collected via telegraf, and the particular metric that is used to detect failed VMs is `uptime`. This metric simply measures how

long a particular VM has been alive for. When `uptime` ceases to be updated for a period of time (current default is 30 minutes) the VM is declared dead and is slotted for destruction. Since within Butler all concerns related to creation and destruction of VMs are handled by terraform it is necessary to ensure that terraform can be executed by the self-healing agent and is kept up to date of the infrastructure state that may be updated elsewhere. Details of the terraform cluster lifecycle management system are described in Section 3.4.1. To support automated relaunching of VMs several changes to how terraform is used are required.

Typically, Butler clusters are launched from a machine that is external to the cloud that is being deployed to. Oftentimes this happens from an individual user's computer. By default terraform creates and keeps up to date a file called `terraform.tfstate` which contains a tree representation of the entire Butler infrastructure. This file typically does not contain sensitive information, thus, it can most frequently be simply checked into source control and checked out on the host that will run the self-healing agent. When it is desired to keep the state information private the data can be encrypted and shared in the manner described below for other private data. Not all of the terraform configuration information is non-private. In order to successfully communicate with cloud API endpoints sensitive configuration settings such as endpoint URLs, paths, and user credentials are required. Relaying this information over to the host running the self-healing agent in a secure manner is handled via passing of a GPG encrypted variable through saltstack.

Under this scheme a public/private GPG keypair is created and the public key is imported on the machine that is holding the secret data and will perform the encryption. The private key is imported on the salt-master machine of the Butler cluster. Secret data is encrypted and stored as a string literal in a salt pillar variable that can be checked into source control, see Listing 3.

Listing 3: Storing GPG encrypted data in a salt pillar variable.

```
1 #!yaml/gpg
2 tf_vars: |
3     -----BEGIN PGP MESSAGE-----
4
5 hQEMAOLYc5Mk0CPHAQf8CeN7ykRp518Fm3co0DF5q8pwk9092ctmDnAhhAascZ2c
6 3QESmEIswLVWhKjbZ9tqmm0JeBR+i5gadJjeMStGLHjJm4hNeWLvduk9y63Vvh07
7 sjm1N+zzEeYINo j5dkFn9ursggwVP/yk7t1KovvhC06vv0dIh3UI1S+BzfYb79Sd
8 aQGMID1n2LoYCy2y1xXjpT1sYU9uVPhQds2WeFL3Kfhc9B8Q/5i58XdiISVg8ueo
9 pg98u00tIe9BruB6m6tRG6f1W3ZDhjpoBE+DrcBm5k8LZ1khZV70+SjTuoMNOdD1
10 K84meNBD0ATi5x0FpfNLnQWXLjJkxaxWaZdGaBplhdLAJAF/PvyvMG0BX7XJdCcW
11 XGs3BoM+NUJqDgjI5gytmLbPRHA2YWUVNTBmUUw6r9abJyBta1w2RwOFMxEGnszL
12 kTOkBZCAoWiCkgug2G5mnwwP9Wh/CERDCuq0bPKHdnES0EMkZ6Bpo7cwX2HMUNj
13 5NAYt0gJ6uD1n3zGstQ8Crchj7rkDqottQ/b1JkgFoXKGCYLnv9EuZxaRXz1ad0n
14 V7We5/GI+4PmemBLKOHQsdaE/z9sveF1xwY1iUb9hHrHZL2s68nJ4T4VU6VSfVZa
15 9W1p4QHic5VDacisq/gWYbSL/ERqKA==
16 =B+ao
17 -----END PGP MESSAGE-----
18
19 terraform_files: /opt/eosc_pilot/deployment/embassy/
```

```

20 terraform_state: /opt/eosc_pilot/deployment/embassy/terraform.tfstate
21 terraform_vars: /opt/eosc_pilot/deployment/embassy/ebi_credentials.tfvars
22 terraform_provider: openstack

```

When the pillar variable is targeted to the VM that runs the self-healing agent and needs to be able to execute terraform commands, the salt-master uses its copy of the private key to decrypt the file contents and deliver them to the appropriate VM as a file (see Listing 4) that is ready to be consumed by terraform.

Listing 4: Retrieving contents of a GPG encrypted pillar variable and storing them in a file.

```

1 {{ pillar['terraform_vars'] }}:
2   file.managed:
3     - contents_pillar: tf_vars
4     - user: root
5     - group: root
6     - mode: 600
7     - makedirs: True

```

Because of the ephemeral nature of cloud-based VMs it is not advisable to simply store terraform state in a file as may be sufficient on a user’s computer. Section 3.7.5 describes Consul, which is a software package that handles Service Discovery concerns in Butler, but also provides a distributed key-value store. In order to support a more robust storage back-end for terraform state we ingest this state data into a Consul key-value store, which terraform can subsequently interact with directly. This requires setting some terraform initialization parameters to indicate a Consul back-end and a location of the terraform state file to ingest. Because the key-value store is replicated across several Consul nodes it is robust to individual VM outages and supports the deployment of several self-healing agents against one back-end as may be necessary for scalability.

Listing 24 shows self-healing agent code excerpts that demonstrate the general flow of terminating and relaunching of a worker VM. Each machine managed by saltstack exchanges SSH keys with the salt-master. To ensure that there are no collisions between keys when reusing worker machine names the defunct worker’s SSH key needs to be removed from the salt-master. Subsequently, terraform’s `taint` command is used to mark the VM for destruction. Once that succeeds the terraform `apply` command is used to actually carry out the destruction and re-provisioning of the VM using an appropriate cloud-provider-specific set of instructions. Since the configuration of the newly launched VM cannot continue until the machine is registered with the salt-master the agent periodically polls salt-master until it is able to successfully retrieve the new key. Once the new key is available configuration proceeds according to the same recipe as when manually launching the Butler cluster. First, DNS is configured via dnsmasq, then service lookup is configured via consul, and lastly the full salt highstate is executed to install the rest of the necessary software. Upon the completion of this process the newly launched VM is placed in service and can take on new tasks from the scheduler.

Self-Healing Caveats

Self-healing is a powerful concept and can play a huge role in facilitating the smooth operation of large computational clusters with minimal human intervention. It is possible though, to over-automate, ending up in a situation where rules are frequently triggered under false conditions, causing needless restarts and wasted system resources. This is due to the sheer number of different things that can go wrong in a large-scale distributed system and the relative difficulty of expressing the resultant conditions and resolution steps in a manner that isn't prone to false positive signals and low robustness. Thus, it is imperative for the Butler user to approach self-healing as a component of the system that requires tuning, project-to-project and environment-to-environment.

When deciding which aspects of the system operational behaviour to automate, the user should keep in mind the level of regularity with which particular classes of issues occur, the reliability of the metrics signal in detecting these issues, and the potential benefits of automated action versus the potential costs of acting on false-positive signals. This can require getting detailed knowledge of the operational profile of the individual algorithms that are being executed as part of a particular project in order to understand what normal and unnormal operating conditions look like so that these may be encoded as anomaly detection rules. This is most easily accomplished by careful observation and analysis of monitoring dashboards to identify important patterns. With the self-healing system it is advised to start with a small number of robust signals, and build up the rule complexity as the users' level of understanding and sophistication with the Butler framework increases in order to reap the most benefits.

3.7.5 Service Discovery

The Butler framework consists of many different services that reside on a number of different servers and need to be able to communicate with each other. To accomplish this in a flexible manner we need to establish a Service Registry so that IP addresses of servers that host particular services can be looked up by service name. To accomplish this Butler uses an Open Source service discovery framework called Consul[47].

Consul provides a cross-data-center distributed Service Name Registry that is available via HTTP and DNS protocols. In addition to registry capabilities Consul provides basic health checks for the underlying services, testing whether the IP and port the service is supposed to be listening on are actually reachable.

Each VM in the cluster runs a Consul Agent which can be run in Server and Client modes. The set of Consul Servers form a Raft cluster and provide consensus-based responses to service lookup requests from clients. When new VMs are started they need to join the Consul cluster in order to be able to perform lookups, doing so requires knowing the IP address of at least one server. In Butler we use Saltstack's configuration capabilities to convey a Consul Server's IP address to any new VM that is brought up.

Registering a service with Consul is a matter of creating a JSON formatted configuration file that declares the service (see Listing 25). A Consul agent running on that VM will ingest the service definition and relay it to a Consul Server which will then circulate it amongst the other servers. The service will then

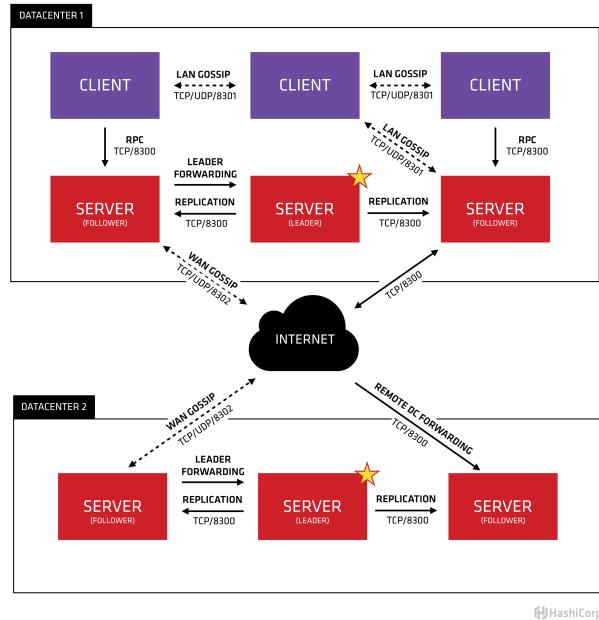


Figure 3.32: Consul high level architecture (from <https://www.consul.io/docs/internals/architecture.html>).

be available for DNS lookups. The service from Listing 25 will have the name `postgresql.service.consul`, for example.

Chapter 4

The Butler Framework - Implementation and Experimental Validation

The Butler framework has been built to enable large-scale scientific analyses in the cloud and the largest set of analyses that have been performed using this framework to date have been the Germline analyses for the Pan-Cancer Analysis of Whole Genomes Project on the EBI Embassy Cloud. We describe the nature of these analyses as well as the details of the Butler deployment on the Embassy Cloud next.

4.1 Deployment on EMBL/EBI Embassy Cloud

The EMBL/EBI Embassy Cloud[42] is an academic cloud computing center which runs an Openstack-based environment. The Embassy Cloud plays a key role in the PCAWG project by donating substantial storage and cloud computing capacity over the course of 3 years. The total amount of resources dedicated to the project by the Embassy Cloud is:

- 1 PB Isilon storage shared over NFS
- 1500 compute cores
- 6 TB RAM
- 40 TB local SSD storage
- 10 Gb network

These resources have been used to host one of the seven PCAWG GNOS repositories that exist worldwide, as well as performing a number of scientific analyses for the project. We have used Butler extensively on the Embassy Cloud in order to carry out the germline analyses for PCAWG-8 Working Group.

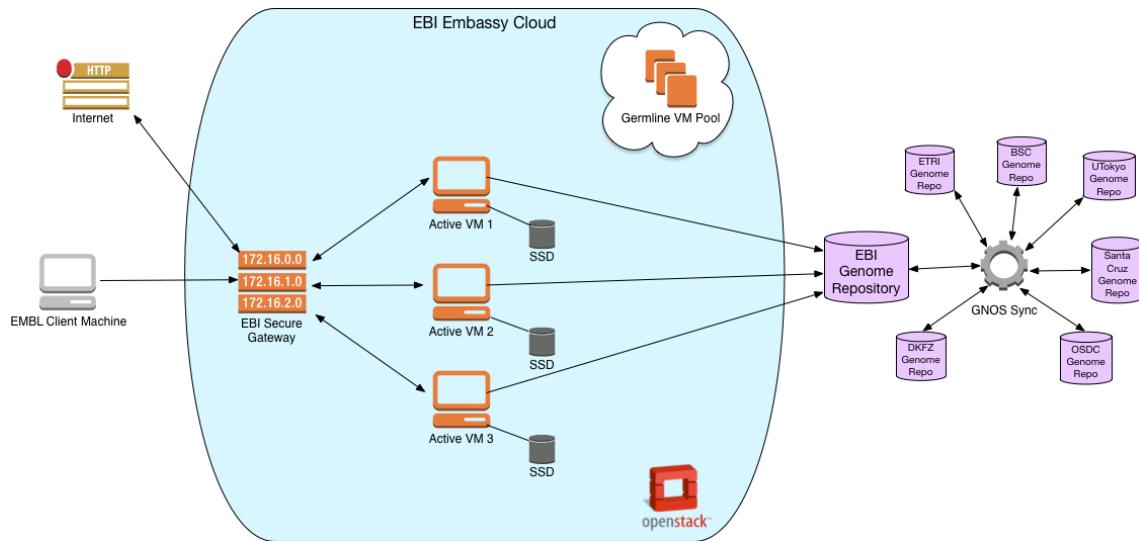


Figure 4.1: Embassy Cloud Architecture

Embassy Cloud Architecture

Figure 4.1 shows the general high-level architecture of the Germline Working Group's tenant within the EMBL/EBI's Embassy Cloud. Because of the sensitive nature of the genetic data that is stored at Embassy there are several security mechanisms in place. The Virtual Machines are hidden behind a secure gateway and are not visible to the external network. The secure gateway maintains a hand-curated list of IP addresses that are allowed to connect to it from the Internet. Currently this list contains several IP addresses of research institutions that are part of the PCAWG project. Beyond the gateway is a bastion host - a Virtual Machine which serves as the entry-point into the cloud environment. Individual users can establish SSH sessions to the bastion host using their SSH key. From the bastion host the user can establish key-based SSH access to other Virtual Machines within the tenancy. Authenticated Web-based access to the Openstack dashboard (Figure 4.2) provides a conventional method for the users to create and manage Virtual Machines.

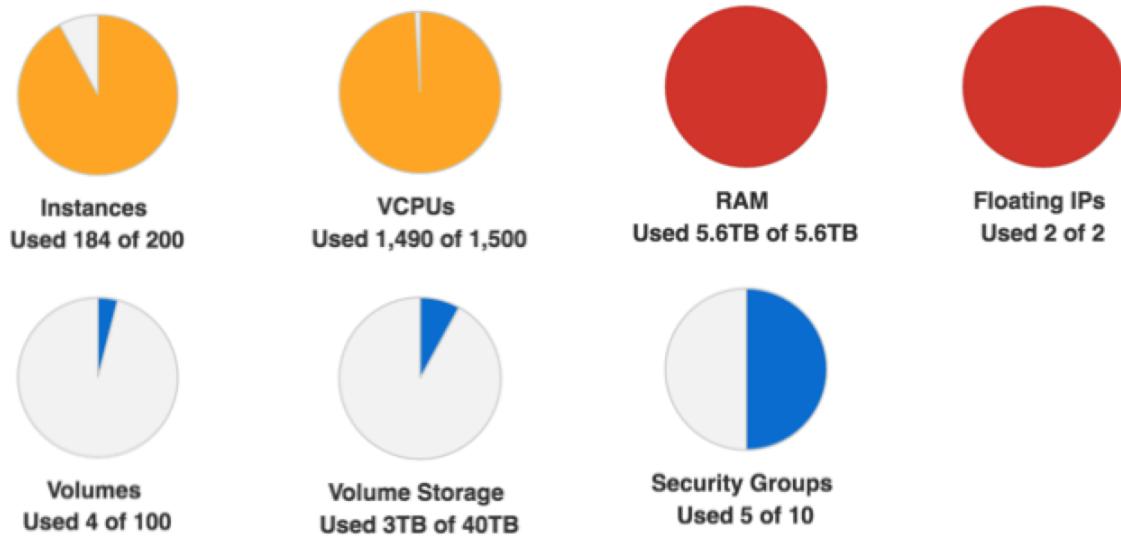


Figure 4.2: Embassy Cloud Dashboard

Access to PCAWG Data

The raw data for PCAWG is hosted in a distributed manner in GNOS repositories. A data synchronization mechanism copies data between repositories when necessary. The EMBL/EBI GNOS repository is one of the most complete sources of PCAWG data, hosting close to 1PB of data for the project. Although typically access to the GNOS repository is only available via a GNOS client the Embassy Cloud IT team has made a special provision for the Germline Working Group to allow access to the underlying data via an NFS share. This allows Butler-based workflows to have more efficient access to the data.

The PCAWG project periodically publishes an official list of all samples that are part of the project. In order to facilitate accurate sample tracking for analysis purposes we have built a Sample Tracking Database on top of PostgreSQL and SQLAlchemy. There are two tables `pcawg_samples` and `sample_locations`. `pcawg_samples` maintains a list of official samples along with their accompanying metadata while `sample_locations` contains a set of file paths that indicate where to find each sample on the Embassy Cloud file server. This table is populated by a script that crawls the directory structure looking for samples that are part of the official list.

Butler deployment

Butler has been deployed on the Embassy Cloud since March, 2016 and has been used extensively to carry out analyses for the Germline Working Group.

To deploy Butler on the 1500 core cluster we set up five different profiles of VMs, each playing a number of different roles (Table 4.1):

Machine	CPU	RAM (GB)	Disk (GB)	Roles	Count
salt-master	4	6	50 ephemeral 1000 block for metrics storage	salt-master consul-bootstrap monitoring-server	1
tracker	4	4	40 ephemeral 1000 block for elasticsearch	tracker consul-server elasticsearch	1
job-queue	4	4	40 ephemeral	job-queue consul-client	1
db-server	8	16	80 ephemeral 1000 block for db	db-server consul-client	1
worker	8	32	100 ephemeral	worker germline consul-client	175

Table 4.1: Butler deployment on Embassy Cloud

Each profile is defined separately via Terraform and uses Saltstack roles for config-

uration. The user can check out the Butler github repository to their local machine and once they install Terraform locally, and provided that they are able to connect to the EBI Secure Gateway (Figure 4.1) they can fully commandeer the provisioning process from the local machine via Terraform.

Figure 4.3 provides a diagrammatic view of the deployment of various Butler components on the Embassy Cloud.

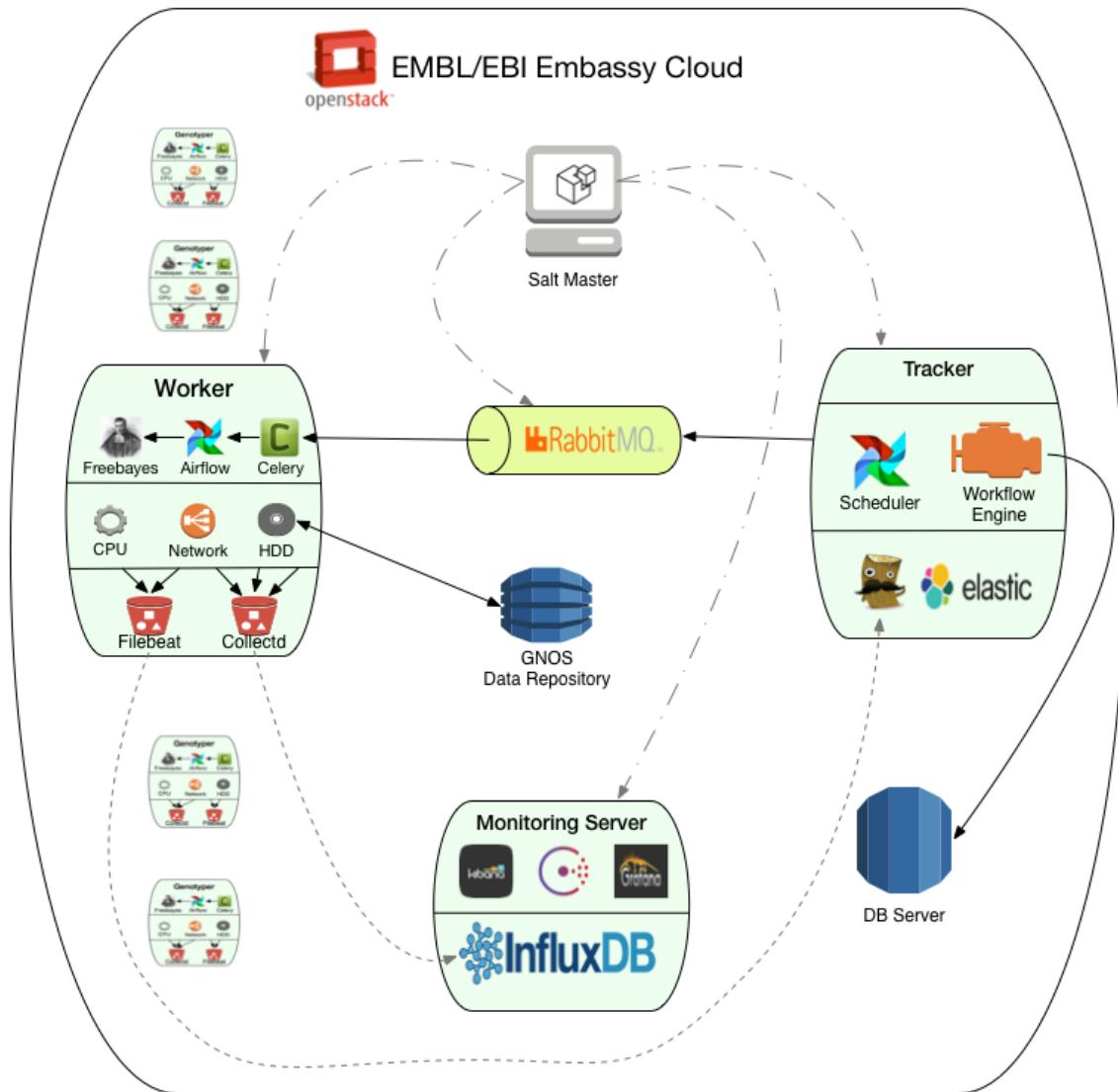


Figure 4.3: Butler Deployment Architecture

The cluster is bootstrapped via the salt-master VM. This VM is started first whenever the cluster needs to be recreated from scratch. The salt-master is started with a minimal OS image for speed and all of the other configurations are delivered via Saltstack itself. The IP of the salt-master machine is retained so that it can be passed on to the other VMs upon creation so that they know how to connect to the master when they boot up. The salt-master VM plays two other roles (Table 4.1) in this deployment in order to maximize resource utilization (since Saltstack is a light resource consumer) - consul-bootstrap, and monitoring-server. The consul-bootstrap role conveys the responsibility for starting up the Consul Service Discovery mechanism to the salt-master. When set up in bootstrap mode, consul waits for one

more consul server to join the cluster, before quorum is reached and the cluster becomes fully operational. The monitoring-server role is responsible for installing and configuring InfluxDB and other monitoring components as well as registering them with Consul so that metrics can start being recorded. We also attach a 1TB block storage volume for the metrics database so that it can survive cluster crashes and tear-downs. If the monitoring server needs to be recreated, the block storage volume simply needs to be reattached to the new Monitoring Server VM.

The tracker VM is responsible for running various Airflow components such as the - Scheduler, Webserver, and Flower (Section 3.6). Additionally we deploy the Butler tracker module (Section 3.6.3)to this VM, thus the tracker VM acts as the main control point of the system from which analyses are launched and monitored. This VM additionally has the elasticsearch role which designates it as the location of the Logstash and Elasticsearch components (Section 3.7.3). To persist the search index we attach an additional 1TB block storage volume. The consul-server role allows the cluster, once the tracker VM is brought up, to reach quorum necessary for full Consul functionality.

The job-queue VM is responsible for hosting the RabbitMQ server which holds all of the in-flight workflow tasks. Because the resources of the job-queue are heavily taxed by communication with all of the worker VMs in the cluster we do not assign any additional roles to this host.

The db-server is responsible for hosting most of the databases used by Butler. This VM runs an instance of PostgreSQL Server and hosts the Run Tracking DB, Airflow DB, and Sample Tracking DB. The 1TB block storage volume serves as the backing storage mechanism.

The worker VMs are the workhorses of the Butler cluster. In its current deployment (October 2016) there are 175 8-core worker machines that are dedicated to running Butler workflows. The worker role ensures that Airflow client modules are installed and loaded on each worker. The germline role additionally loads the workflows and analyses that are relevant to the PCAWG Germline Working Group.

4.2 PCAWG Germline Analyses

The PCAWG project is divided into a set of working groups. Each group has a different set of research interests and technical activities that it is contributing to the overall project effort. The goal of the Germline Working Group, also known as PCAWG-8, is to study the distribution of germline (mutations that are inherited from one's parents) polymorphisms within the PCAWG cohort of 2834 cancer patients and gain a better understanding of how these germline polymorphisms affect various aspects of the patients' disease, for instance whether they affect disease progression, likelihood of survival, or any number of molecular-level traits such as DNA repair, propensity towards certain types of mutations, or gene dysregulation.

To enable these analyses the goal of the Germline Working Group is to produce a full set of high quality genotyped germline variants. Doing so requires carrying out a significant number of computational steps that use the entire 725 TB raw data set. These steps are as follows:

Variant Discovery - Use a set of algorithms that look at each location in the genome and try to determine where the genome differs from the known refer-

ence sequence.

Variant Genotyping - Using a set of variant sites produced by Variant Discovery and determine an accurate genotype at the variant position for all donors in the cohort.

Variant Filtration - Filter out false positive calls introduced by the previous steps.

Genotype Phasing - Use an algorithm to determine which chromosome (of the pair) each variant belongs to.

Data Submission - Prepare metadata and submit the resulting call-set to a centralized data repository.

4.2.1 Variant Discovery

There exist multiple algorithms for variant discovery and each algorithm has a unique set of features. As a result, they typically produce call-sets that only overlap on a subset of the values[129]. In order to improve the sensitivity of the call-set the Germline Working Group is producing three independent discovery call sets via three different algorithms - Freebayes[78], GATK HaplotypeCaller[54], and RTG[33]. These call-sets are then merged via a two-out-of-three criterion i.e. a variant is retained if it is called by at least 2 of the three pipelines. This approach produces a more sensitive call-set than via any of the tools individually.

The GATK HaplotypeCaller data set has been produced by the Broad Institute, the RTG set has been produced by Stanford University, and the Freebayes data set has been produced using a Butler workflow on the EBI Embassy Cloud.

The freebayes Butler workflow

The freebayes workflow parallelizes its work by splitting each sample by chromosome to reduce the amount of time it takes to process a single sample. Although the chromosomes have vastly different sizes (see Table ??), and thus individual jobs have different runtimes, when many samples are processed, there is little practical impact on when the entire batch of samples is completed.

Chromosome	Size(base pairs)
1	249,250,621
2	243,199,373
3	198,022,430
4	191,154,276
5	180,915,260
6	171,115,067
7	159,138,663
8	146,364,022
9	141,213,431
10	135,534,747
11	135,006,516
12	133,851,895
13	115,169,878
14	107,349,540
15	102,531,392
16	90,354,753
17	81,195,210
18	78,077,248
19	59,128,983
20	63,025,520
21	48,129,895
22	51,304,566
X	155,270,560
Y	59,373,566

Table 4.2: Human chromosome size distribution

Overall, most Butler workflows that carry out an analysis follow a similar structure - an Analysis Run is started, access to the sample is validated, the analysis steps are carried out (possibly with branching), and the Analysis Run is completed (see Figure 4.4).

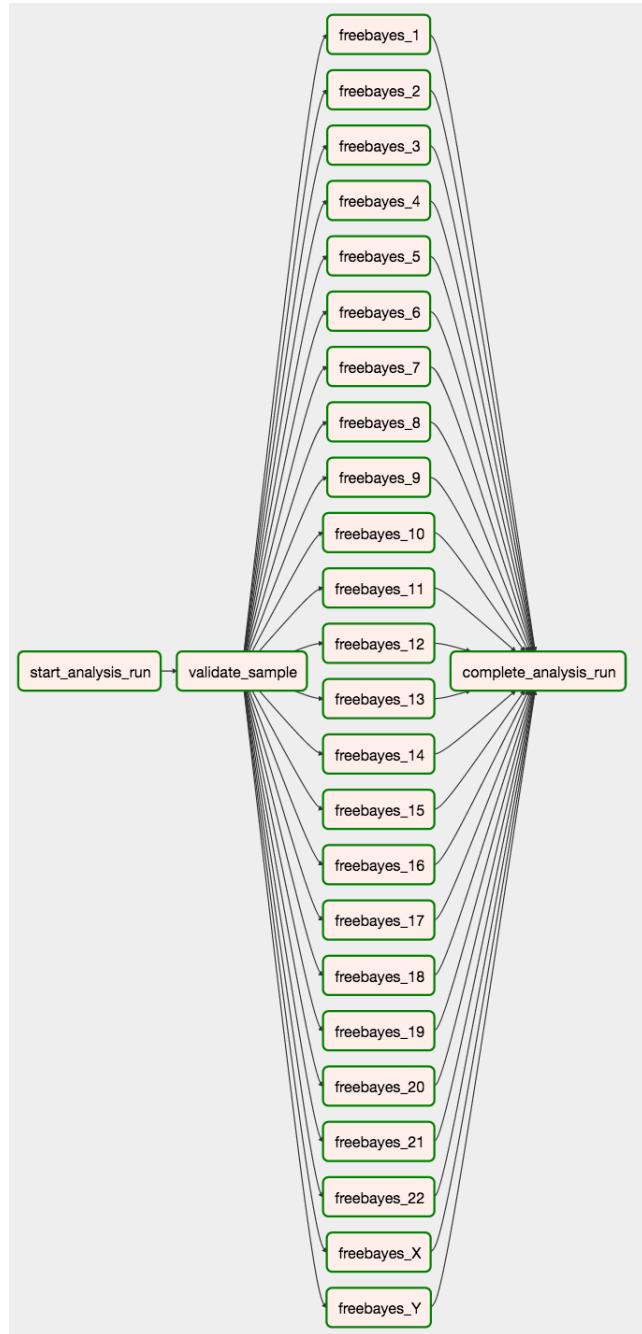


Figure 4.4: Structure of the Butler freebayes workflow

Because of the largely common structure between workflows a large degree of code reuse is possible, thus most of the methods reside in the `workflow_common` submodule of the tracker module (see Section 3.6.3) and are invoked for each workflow. A full listing of the source code for the freebayes workflow is provided in Listing 26 and is discussed next.

Lines 81-127 of the source code define the workflow structure, first by declaring an instance of type `DAG`, and then by defining a sequence of workflow tasks. In this case each task is a Python callable. The loop on line 117 defines one workflow task for each chromosome in the predefined list. The order of task execution is defined by calling a task's `set_upstream()` method, such as on lines 109 and 127 of the listing. Default parallelism behaviour is specified on line 92 where the maximum number

of active workflow runs is defined to be 2000, and the maximum number of active workflow tasks is defined to be 10,000. If more workflows than the maximum get scheduled, they will be queued until some workflow instances complete.

The bulk of the body of the workflow definition (lines 14-78) is dedicated to the implementation of a single function - `run_freebayes(**kwargs)` which manages the invocation of the freebayes tool on a single chromosome of a sample. Line 16 gets the effective configuration dictionary (see Section 3.6.4) which contains the merged configuration parameters from Workflow (Listing 5), Analysis (Listing 6), and Analysis Run (Listing 7) levels.

Listing 5: Workflow-level configuration for freebayes workflow.

```
1  {
2      "contig_names": ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
3                      "11", "12", "13", "14", "15", "16", "17", "18", "19",
4                      "20", "21", "22", "X", "Y"],
5      "reference_location": "/reference/genome.fa",
6      "bgzip": {
7          "path": "/usr/local/bin/bgzip",
8          "flags": ""
9      },
10     "tabix": {
11         "path": "/usr/local/bin/tabix",
12         "flags": "-f -p vcf"
13     },
14     "rsync": {
15         "flags": "-a -v --remove-source-files"
16     },
17     "freebayes": {
18         "path": "/bin/freebayes"
19     }
20 }
```

The workflow-level configurations contain values that should generally be applicable to any invocation of the workflow. In exceptional cases these can be overridden at Analysis and Analysis Run levels. For the freebayes workflow these settings include a list of chromosomes to call, the path to the human reference genome, and paths to various tools used within the workflow

Listing 6: Analysis-level configuration for freebayes variant discovery analysis.

```
1  {
2      "results_base_path": "/shared/data/results/discovery/",
3      "results_local_path": "/tmp/discovery/",
4      "freebayes": {
5          "mode": "discovery",
6          "flags": "--min-repeat-entropy 1
7                  --report-genotype-likelihood-max"
8      }
9  }
```

```

7     }
8 }
```

Since our analysis focuses on variant discovery, the Analysis-level JSON configuration file contains freebayes flags to set up discovery mode, as well as setting up a location for where to store the analysis results and which directory to use as local scratch space.

Listing 7: AnalysisRun-level configuration for a single sample in freebayes variant discovery analysis.

```

1 {
2     "sample": {
3         "sample_location": "/gnosdata/tcga/PCAWG.67455c36-aa47-4cc4-8b6d-
4             ↳ 9a9012b616ed.bam",
5         "donor_index": 0,
6         "sample_id": "f22a72c5-73c8-478d-b03e-04599b9d5321"
7     }
}
```

Listing 7 provides an example of what an AnalysisRun-level configuration looks like. This configuration is concerned with supplying sample level configuration values, such as the sample_id and sample_location.

After all of the necessary parameters are extracted from the configuration and command invocation is set up lines 71-75 of Listing 26 actually invoke a series of commands that perform the bulk of the analysis - calling `freebayes` to generate the discovery call-set, followed by converting the call-set into a binary compressed format (with `bgzip`), followed by generating an index file for record-based random access into the binary file (with `tabix`), and followed by an `rsync` to the shared results storage indicated in the configuration.

The workflow is distributed to all worker nodes in the cluster via a Saltstack state as shown in Listing 27.

AnalysisRun configurations for freebayes workflow

While each workflow only has one workflow-level configuration and possibly a few dozen analysis-level configurations, there needs to be one analysis run-level configuration generated for each sample under analysis, thus resulting in thousands of these configurations being generated for each analysis. The most effective method for accomplishing this is via a script. We utilize two databases - the Run Tracking Database (Section 3.6.3), and the Sample Tracking Database (Section 4.1) in order to generate a list of samples for which there are no Analysis Runs present for a given Analysis yet. To generate our result-set we utilize the SQLAlchemy Object-Relational Mapping framework (see ??).

Listing 8: SQLAlchemy query to generate available samples.

```
1 current_runs = run_session.query(Configuration.config[("sample", "
2   ↳ sample_id")]).astext().\
3     join(AnalysisRun, AnalysisRun.config_id == Configuration.config_id).\
4     join(Analysis, Analysis.analysis_id == AnalysisRun.analysis_id).\
5     filter(and_(Analysis.analysis_id == analysis_id, AnalysisRun.run_status
6       ↳ != tracker.model.analysis_run.RUN_STATUS_ERROR)).all()
7
8 available_samples = sample_session.query(PCAWGSample.index.label("index"),
9   ↳ sample_id.label("sample_id"), sample_location.label("sample_location")).\
10    join(SampleLocation, PCAWGSample.index == SampleLocation.donor_index).\
11    filter(and_(sample_location != None, sample_id.notin_(current_runs))).\
12    limit(num_runs).all()
```

The final script is wrapped in a Command Line Interface to improve the user experience. It supports the following parameters:

analysis_id - The id of the Analysis for which to generate Analysis Run configs.

num_runs - The number of runs to generate. The actual number of runs will be `min(num_runs, available_runs)`

tissue_type - Whether to generate the Analysis Runs for tumor or normal tissue samples.

config_location - File path where to store the resulting Analysis Run configs.

Thus, a full invocation would look like:

```
python prepare_freebayes_genotyping_config.py create-configs -a 3 -n
→ 150 -t normal -c /config_file_location/
```

This would generate at most 150 JSON files with configurations for Analysis ID 3 and normal tissue samples, storing them in `/config_file_location/` which could be used to start workflow instances for this analysis.

4.2.2 Variant Genotyping

Genotyping refers to accurately determining for each sample and at each variant position what are the two nucleotide bases (one for each sister chromosome) at that position[223]. This analysis involves looking at the DNA reads that overlap each position and evaluating a model for the likelihood of each possible genotype given the data observed in the reads. The genotype with the highest likelihood given the data is selected[160]. To accomplish this task we use a Butler workflow that utilizes freebayes as the computational algorithm underneath the covers. Because the freebayes workflow from Section 4.2.1 has been built in a generic fashion the only changes that are necessary between discovery and genotyping analyses lie within the analysis configuration.

We see in Listing 28 that we need to provide a list of variant locations that need to be genotyped, split by chromosome, and stored in VCF format[49]. Additionally, we provide a set of flags to freebayes that indicate that the tool should be used in genotyping mode.

4.2.3 Variant Filtration

Although utilizing multiple variant callers for variant discovery improves the overall sensitivity it also increases the number of false positives in the call-set. In order to remove the false positive calls the Germline Working Group has undertaken a number of filtration steps. Some of these steps involve machine learning methods that are carried out outside the scope of Butler, but some are based on a set of well-known filtering criteria. These are implemented as a separate Butler workflow.



Figure 4.5: Structure of the Butler Variant Filtration workflow

The Filter Variants workflow has a rather simple structure. Bookended by the standard run-start and run-completion tasks is the actual filtration task. This task is implemented as an Airflow PythonOperator and invokes two commands - vcftools[49] and vt[209]. vcftools is used for actual variant filtering, while vt is used for variant normalization.

Listing 9: Butler Analysis configuration for VCF filtering.

```

1  {
2      "results_base_path": "/shared/data/results/discovery_filtered/",
3      "results_local_path": "/tmp/discovery_filtered/",
4      "vcffilter": {
5          "flags": "QUAL > 20 & DP > 3 & QUAL / AO > 2 & SAF > 1 & SAR > 1
6              & RPR > 1 & RPL > 1"
7      },
8      "vt": {
9          "command": "normalize"
10     }
}

```

Listing 9 demonstrates the usage of vcftools' flags to achieve variant filtering for PCAWG.

4.2.4 Genotype Phasing

Because each individual inherits one copy of each chromosome (except for sex chromosomes X and Y) from the mother and one from the father, a variant may lie on one chromosome or the other, or both. It is, thus, important to understand which chromosome each variant lies on in order to inform downstream analyses. This methodology is called statistical phasing and will be carried out by a tool called Shapeit[53] outside of Butler.

4.2.5 Data Submission

Once a call-set for each sample is produced and vetted it needs to be submitted to a centralized data repository so that it can be shared with other researchers on the project. There are seven such data repositories throughout the world, each running a software tool called GNOS[233] from Annai Systems. A submission to GNOS consists of the call-set data accompanied by an XML-formatted metadata file, that describes the submission. GNOS then uses a proprietary torrent-like protocol for secure file uploads. A Butler workflow implements automated sample submissions to GNOS.



Figure 4.6: Structure of the Butler Data Submission workflow

The Data Submission workflow (Figure 4.6) follows a linear sequence of events implemented as Airflow PythonOperators. The data submission is a three step process where the first action is to prepare a sample's accompanying metadata submission, the second is to submit this metadata to a GNOS repository of choice, which generates a manifest in return, and the third step is to upload the actual data to the same repository using the manifest. See Listing 29

In order to be able to successfully submit a sample, the sample, along with its accompanying metadata must be placed in a separate directory whose name is a Universally Unique Identifier (UUID) - this UUID will become an identifier for the submission on the GNOS server. Furthermore, the metadata file - an XML document, must be populated with descriptions of the analysis steps taken to produce the sample. We generate this file in Butler's workflow with the aid of an XML template (see Listing 30 template declaration) and using Python's `etree` module. Once the submission is ready, the actual process of submission is carried out using the `cgsSubmit` tool by Annai Systems. It is important which GNOS repository a sample ends up in as not all repositories have permissions to host all samples. The `destination_repo_mapping` dictionary in Listing 30 maintains a mapping between a sample's project and a corresponding GNOS repository name. Listing 29 provides a further mapping between repository names and repository URLs thus allowing `cgsSubmit` to submit each sample to its corresponding GNOS repository. The output of the `metadata_submit` task is a `manifest.xml` file which is placed in the sample's directory and contains all of the necessary information to enable the upload of the actual data.

The `upload_data` task is responsible for moving the actual data into a designated GNOS repository. This is accomplished using a tool called `gtupload` which implements a torrent-like data upload protocol.

4.2.6 Structural Variant Calling

While the previously described methods are geared towards the detection and genotyping of Single Nucleotide Polymorphisms (SNPs), there are other classes of germline variants within a person's genome. Structural Variants form a broad class

of larger polymorphisms which are typically 50 basepairs or larger in size[208]. There are various types of structural variants, including:

- Deletions
- Inversions
- Duplications
- Translocations

The Germline Working Group is using a tool called Delly[177] to accurately detect and genotype these variants. To enable Delly analyses on the EBI Embassy Cloud we have built a Delly workflow in Butler (Figure 4.7).



Figure 4.7: Structure of the Butler Delly workflow

This workflow has a familiar linear structure with the bulk of the work being done inside the delly_genotype task. Because Delly knows how to work with and output compressed VCF files there is no need to compress and generate indexes like with freebayes. This makes the task code simpler (Listing 31).

After extracting the necessary configuration parameters and setting up the delly execution command, once delly execution finishes, the resulting call-set is copied over to its final location. Control over program behaviour is mostly exercised at the analysis level, where program flags are typically indicated (Listing 32)

4.3 Experimental Runs

Between January and October 2016 Butler has been used extensively to facilitate a number of large scale cancer genomics analyses on behalf of the Germline Working Group of the Pan Cancer Analysis of Whole Genomes Project. The input to these analyses has been a 725 TB data-set of 2834 cancer patients' sequenced DNA samples, and the outputs have been a number of call-sets identifying and genotyping various classes of germline variants in the form of VCF files. All of the computations have been performed on the EMBL/EBI Embassy Cloud - a 1500 core, 6TB of RAM, 1PB of storage, academic cloud running Openstack.

In this section we describe the technical details and characteristics of these experimental runs to establish a measure of Butler's effectiveness in real-life scenarios.

4.3.1 Freebayes Common Variant Genotyping

The Common Variant Genotyping analysis refers to the genotyping within the PCAWG cohort the genomic variants that occur with at least 1% Minor Allele Frequency (MAF) within the 1000 Genomes Project's[38] cohort. This site list consists of 12 million variants that need to be genotyped for each patient - thus requiring genotyping at 34 billion sites.

To accomplish this task we utilize the Butler freebayes workflow in genotyping mode, supplying the following configurations:

Listing 10: Butler Freebayes Workflow analysis configuration for common variants genotyping.

```
1  {
2      "variants_location": {
3          "1": "/1000GP_maf_0.01/ALL.chr1.vcf.gz",
4          "2": "/1000GP_maf_0.01/ALL.chr2.vcf.gz",
5          "3": "/1000GP_maf_0.01/ALL.chr3.vcf.gz",
6          "4": "/1000GP_maf_0.01/ALL.chr4.vcf.gz",
7          "5": "/1000GP_maf_0.01/ALL.chr5.vcf.gz",
8          "6": "/1000GP_maf_0.01/ALL.chr6.vcf.gz",
9          "7": "/1000GP_maf_0.01/ALL.chr7.vcf.gz",
10         "8": "/1000GP_maf_0.01/ALL.chr8.vcf.gz",
11         "9": "/1000GP_maf_0.01/ALL.chr9.vcf.gz",
12         "10": "/1000GP_maf_0.01/ALL.chr10.vcf.gz",
13         "11": "/1000GP_maf_0.01/ALL.chr11.vcf.gz",
14         "12": "/1000GP_maf_0.01/ALL.chr12.vcf.gz",
15         "13": "/1000GP_maf_0.01/ALL.chr13.vcf.gz",
16         "14": "/1000GP_maf_0.01/ALL.chr14.vcf.gz",
17         "15": "/1000GP_maf_0.01/ALL.chr15.vcf.gz",
18         "16": "/1000GP_maf_0.01/ALL.chr16.vcf.gz",
19         "17": "/1000GP_maf_0.01/ALL.chr17.vcf.gz",
20         "18": "/1000GP_maf_0.01/ALL.chr18.vcf.gz",
21         "19": "/1000GP_maf_0.01/ALL.chr19.vcf.gz",
22         "20": "/1000GP_maf_0.01/ALL.chr20.vcf.gz",
23         "21": "/1000GP_maf_0.01/ALL.chr21.vcf.gz",
24         "22": "/1000GP_maf_0.01/ALL.chr22.vcf.gz"
25     },
26     "results_base_path": "/shared/data/results/regenotype_1_percent_maf/",
27     "results_local_path": "/tmp/regenotype_1_percent_maf/",
28     "freebayes": {
29         "mode": "regenotyping",
30         "flags": "-l"
31     }
32 }
```

Figure 4.8 shows a distribution of job runtimes (in minutes) separated by chromosome. The mean runtime is highly correlated with chromosome length (and consequently number of variants), with a Pearson correlation of 0.92.

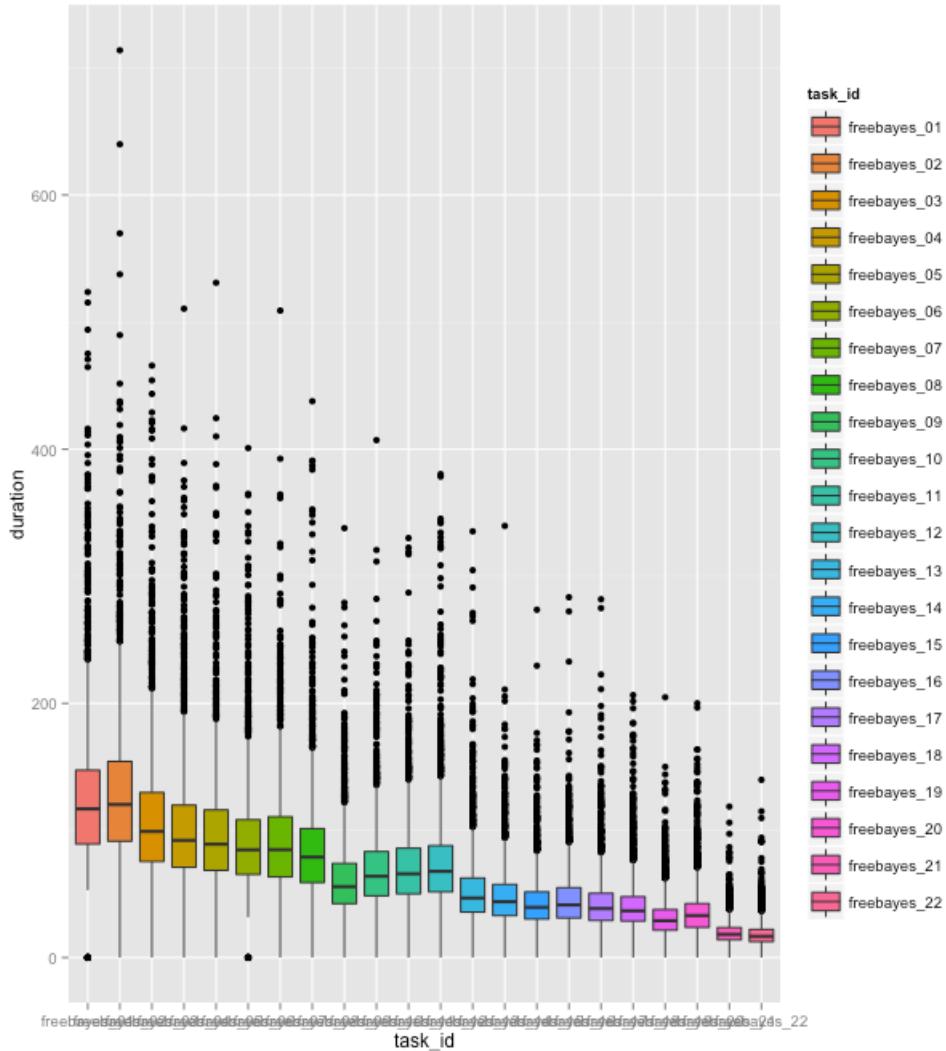


Figure 4.8: Runtimes of freebayes genotyping on the 1% MAF site-list.

Overall 130,152 compute hours were used to complete 70,850 workflow tasks for this analysis with an additional 2688 CPU hours used for cluster management overhead. Thus, management overhead accounted for 2% of the overall computational resource costs for this analysis. Utilizing 1000 cores this analysis took less than 6 days to complete.

4.3.2 Freebayes Variant Genotyping

Using the site-list of 60 million variants obtained from the Freebayes Variant Discovery analysis we used the Butler Freebayes Workflow in genotyping mode to calculate genotypes at 170 billion genomic positions. 76,518 tasks workflow tasks were completed utilizing 302,071 CPU hours over the course of the analysis (10 days wall time), of which 5,040 CPU hours were cluster management overhead, accounting for 1.6% of total resource utilization. Figure 4.9 demonstrates the distribution of task durations by chromosome.

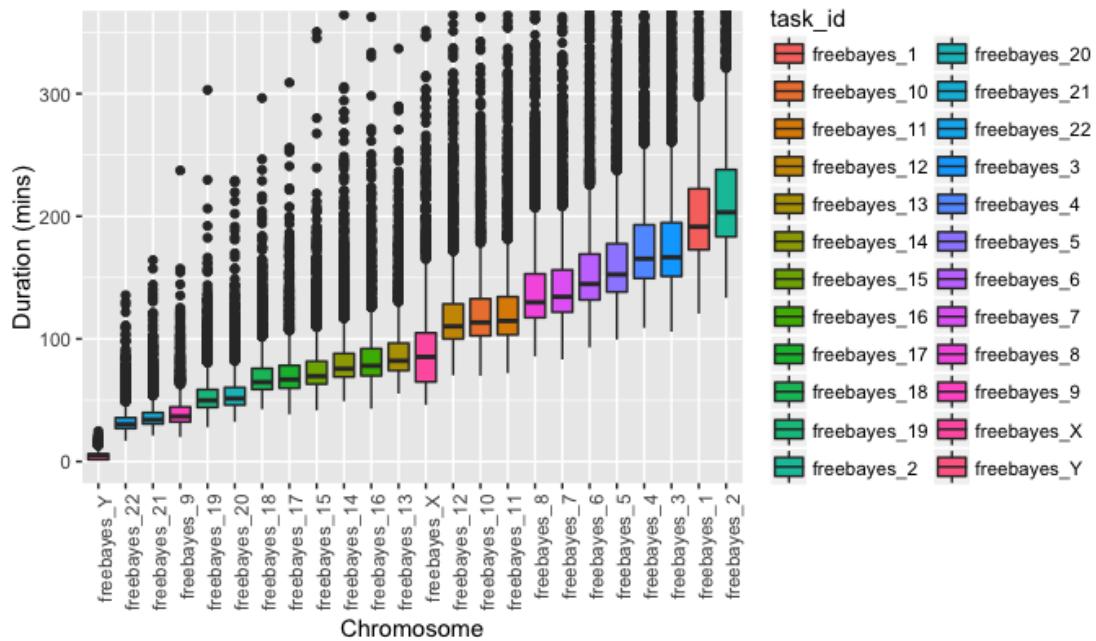


Figure 4.9: Runtimes of freebayes regenotyping on the freebayes discovery call-set.

Figure 4.10 provides a density-based view of task durations split by chromosome. We observe that durations in each case tend to fall about some mean, dependent on chromosome length (Pearson's $r = 0.925$), with variance also decreasing with chromosome length ($r = 0.94$). In each case there is a considerable right tail of duration outcomes, with the maximum duration for each chromosome occurring on average 11.7 standard deviations from the mean.

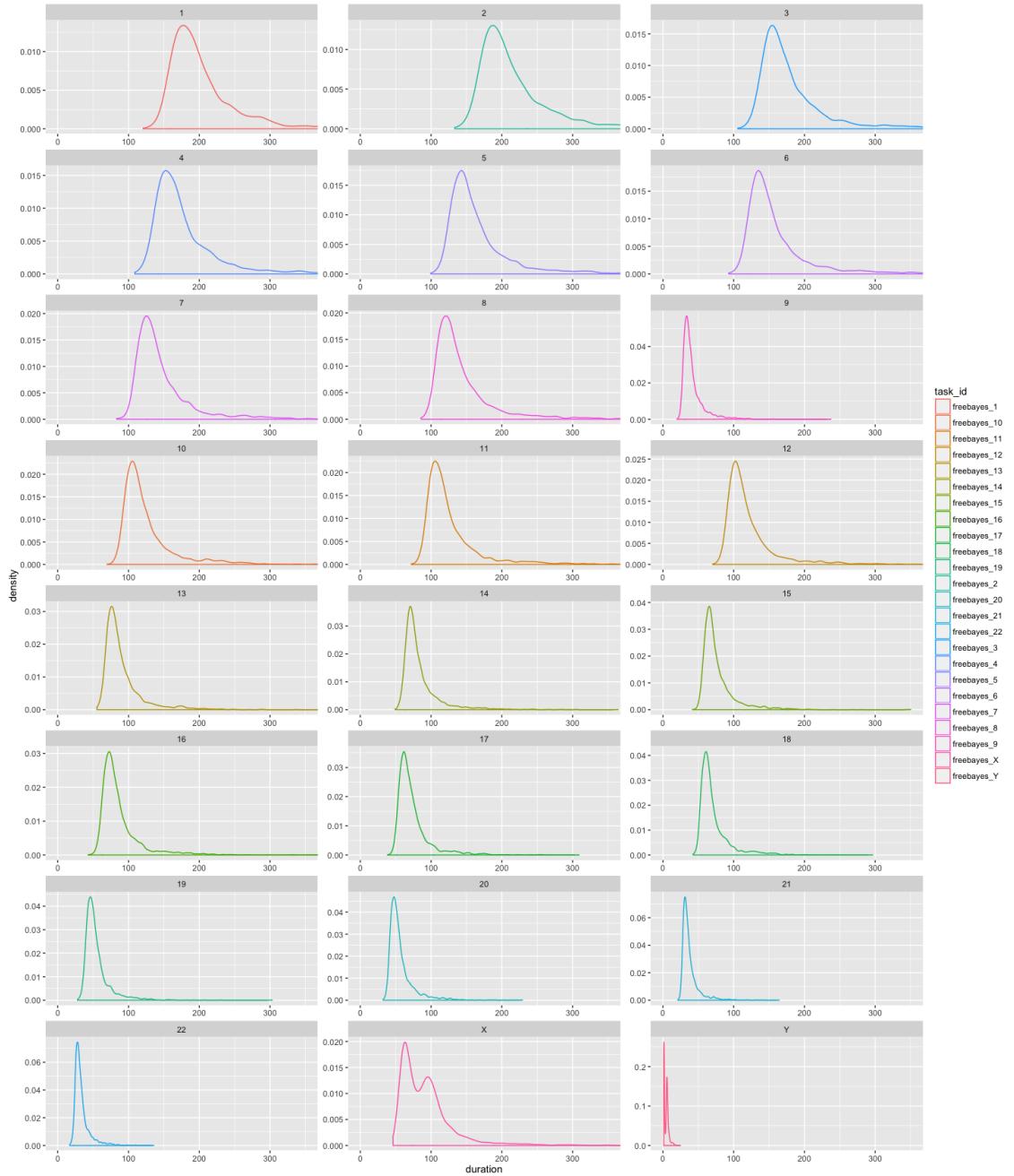


Figure 4.10: Task duration distributions by chromosome.

Figure 4.11 shows a view of the cluster load during the analysis execution. Here we see that overall the load has been stable, with a few sporadic spikes (5/25, 5/28, 5/29). On the other hand we see that the load is not uniform across the cluster with some machines not fully utilized. This is clear from the CPU utilization panel, where the majority of the VMs are at 100% CPU utilization throughout the analysis execution, but several machines appear to be stable at utilization levels between 50% and 90%.



Figure 4.11: Cluster resource utilization during the regenotyping analysis.

4.3.3 Delly Full Variant Genotyping

The analysis of Delly Structural Variant Calling has been split into two parts - genotyping of germline deletions, and genotyping of germline duplications. We consider each in turn.

Deletions Genotyping

The deletions analysis used the following analysis configuration:

Listing 11: Butler Delly Workflow analysis configuration for deletions genotyping.

```

1 {
2     "variants_location": 
3         → "/shared/data/samples/vcf/delly_deletion_sites/del.sites.bcf",
4     "results_base_path": 
5         → "/shared/data/results/delly_germline_deletions_14_07_2016/",
6     "results_local_path": "/tmp/delly_germline_deletions/",
7     "variants_type": "DEL"
}

```

244,889 deletions were evaluated across 5668 samples (tumour and normal) for a total of 1,388,030,852 genomic sites genotyped. Overall wall-time was 13 days, utilizing 265,200 CPU hours with 6240 CPU hours used for cluster management overhead - an overhead of 2.2%.

Figure 4.12 shows a histogram of genotyping (sample level) jobs.

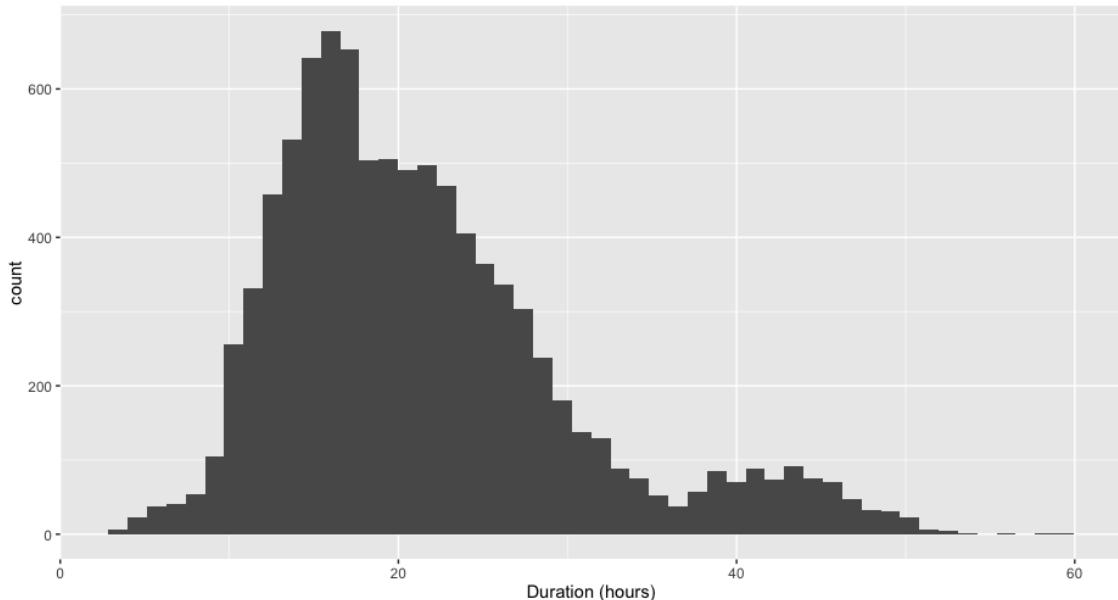


Figure 4.12: Durations of deletion genotyping tasks by sample.

Table 4.3 shows the summary statistics of job durations.

Table 4.3: Summary statistics of delly deletion genotyping job durations

task_id	mean	median	sd	min	max
delly_genotype	21.48	19.70	8.89	3.35	59.30

Figure 4.13 shows the overall cluster load during the deletion genotyping analysis. During this analysis there were several periods during which the Workflow Scheduler failed and the job queue ran out of tasks. These periods can be seen as dips on the Load, CPU, and Memory metrics' graphs.



Figure 4.13: Cluster performance during the deletion genotyping analysis.

Duplications Genotyping

The duplications analysis used the following configuration:

Listing 12: Butler Delly Workflow analysis configuration for duplications genotyping.

```

1  {
2      "variants_location":
3          "→  "/shared/data/samples/vcf/delly_deletion_sites/dup.sites.bcf",
4      "results_base_path":
5          "→  "/shared/data/results/delly_germline_dups_05_09_2016/",
6      "results_local_path": "/tmp/delly_germline_dups/",
```

```

5   "variants_type": "DUP"
6
7 }
```

Overall 217,433 duplications were genotyped for each sample, across 5668 samples for a total of 1,232,410,244 genomic variants genotyped. The wall-time for this analysis was only 4.5 days, utilizing 151,200 CPU hours during this time, with a management overhead of 2160 hours, for a total overhead of 1.4%. The comparatively lower cluster management overhead has been accomplished by scaling up the cluster to 1400 cores without the need for more management resources.

Figure 4.14 shows a histogram of genotyping job durations.

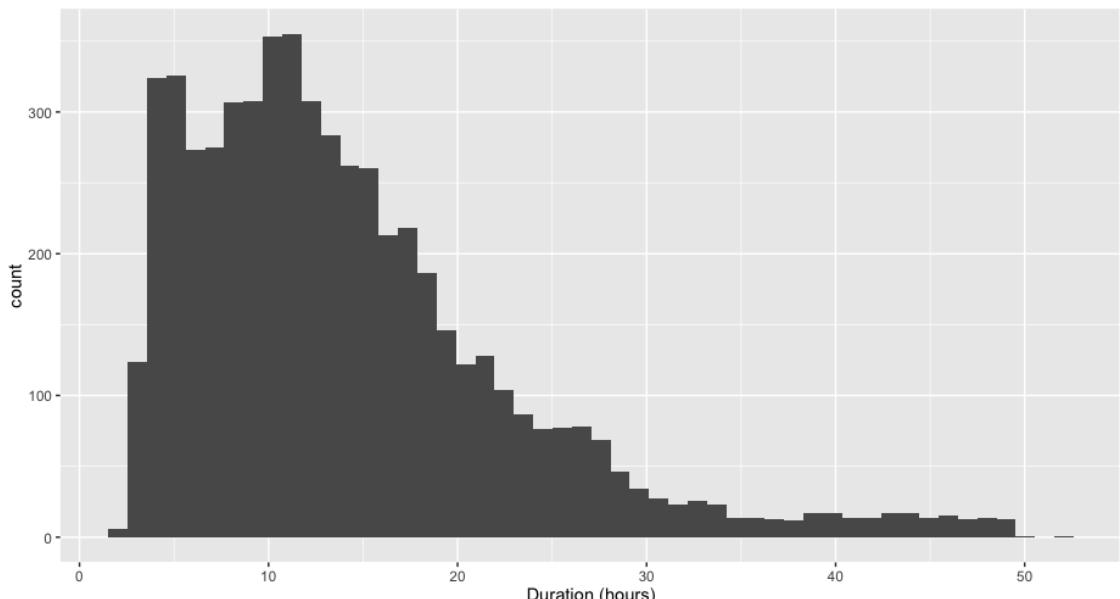


Figure 4.14: Durations of duplication genotyping tasks by sample.

Table 4.4 provides summary statistics of the same data.

Table 4.4: Summary statistics of delly duplication genotyping job durations

task_id	mean	median	sd	min	max
1 delly_genotype	14.27	12.32	8.80	2.15	52.19

Figure 4.15 shows a measurement of cluster performance during the duplication genotyping analysis. This analysis appears to have run very smoothly, with two tranches of data - the normal genomes, and the tumour genomes closely following each other and exhibiting stable Load and Memory performance, and a CPU load profile that is, although spiky, is normal for Delly execution.

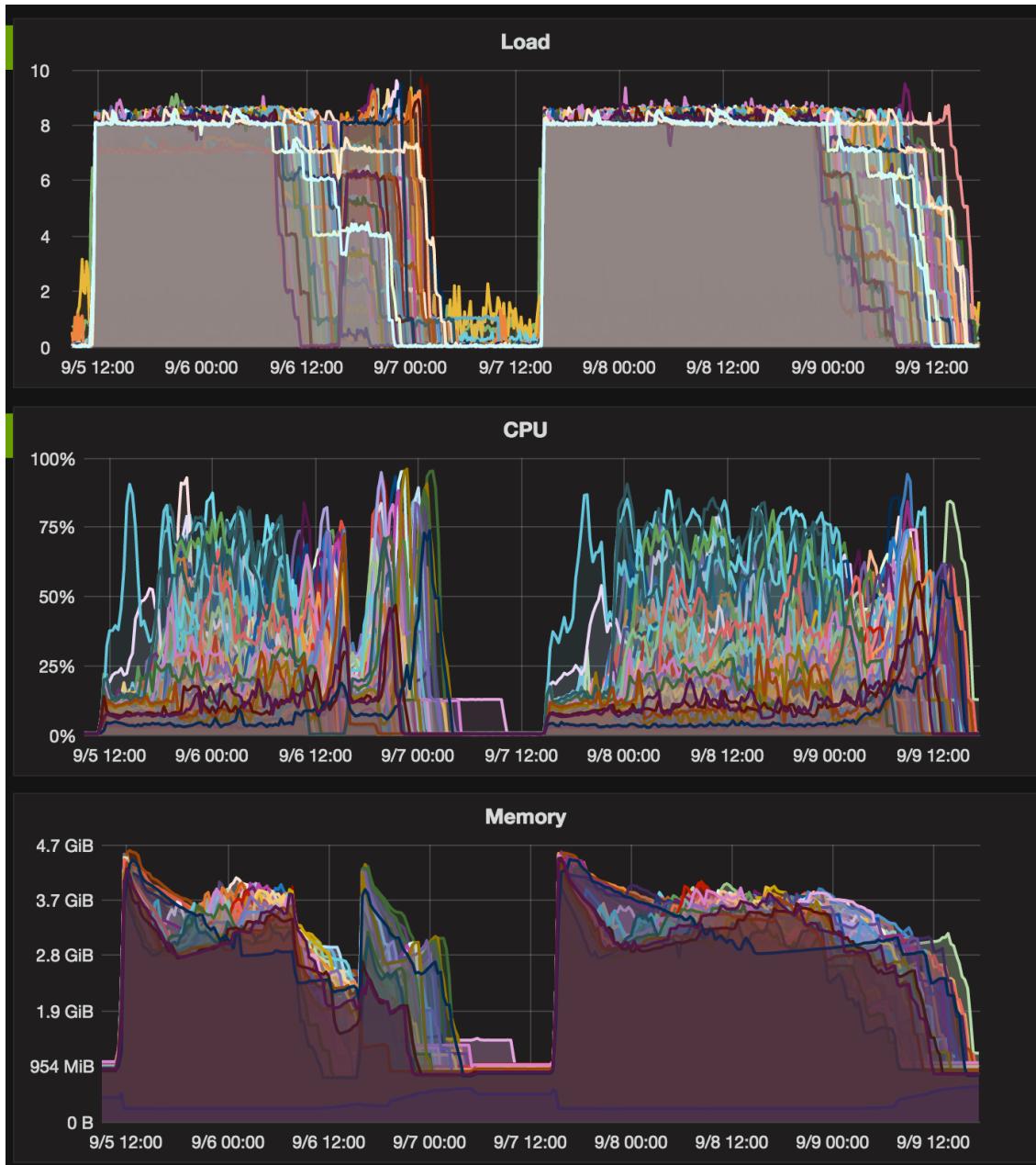


Figure 4.15: Cluster performance during the duplication genotyping analysis.

4.4 Performance Evaluation and Comparison

We evaluate the relative effectiveness of Butler-based pipelines compared to a set of pipelines operating under similar conditions, and over the same data-set, namely the “core” PCAWG somatic pipelines that have been used to accomplish genome alignment and somatic variant calling for the PCAWG Technical Working Group¹. The core PCAWG pipeline set consists of five pipelines – BWA, Sanger, Broad, DKFZ/EMBL, and “OxoG detection” run over the course of 18 months over all samples in PCAWG. The Butler-based pipeline set consists of two pipelines – Freebayes, and Delly, used to accomplish four analyses – germline SNV discovery, germline SNV genotyping, germline SV deletion genotyping, and germline SV duplication genotyping, also running over all samples in PCAWG (725 TB in total). We assessed and

compared pipeline performance with respect to an estimated optimal performance (based on available hardware), as well as with respect to analysis progress uniformity in time.

For core PCAWG pipelines, we used the date of data upload to the official data repository as the most reliable sample completion date. However, approximately 25% of the DKFZ/EMBL pipeline results were uploaded in two batches on two separate days, and thus do not accurately represent the real analysis progress rate. For this reason, we excluded this pipeline from the optimal performance analysis. Butler sample completion dates are based on timestamps collected in Butler’s analysis tracking database.

Our assessment of pipeline performance is based on establishing an “optimal” progress rate for a pipeline given a hardware allocation. We divided the sample set into 20 bins and defined the optimal progress rate for each pipeline to be the smallest proportion of overall analysis time required to process 5% of all samples (scaled to a 1% rate).

$$r_{opt} = \min_{b \in bins} \{duration_b/duration_{total}/5\} \quad (4.1)$$

We observed that the mean r_{opt} was significantly higher for Butler-based pipelines at 0.46 than for the core PCAWG pipelines at 0.13 (See Table 4.5).

Core PCAWG	
BWA	0.16
Broad	0.14
Sanger	0.07
OxoG	0.14

Butler	
SNV Discovery	0.27
SNV Genotyping	0.90
SV Duplications	0.39
SV Deletions	0.28

Table 4.5: Optimal rates for PCAWG and Butler pipelines.

For each pipeline and each 1% of the samples under analysis, we then computed a metric e (for effectiveness) defined as the proportion of r_{opt} actually achieved.

$$e = \frac{r_{act}}{r_{opt}} \quad (4.2)$$

Comparing the core PCAWG and Butler pipelines with respect to e (Figure 4.16) we observed that effectiveness was on average lower for PCAWG pipelines ($\mu_{e_{PCWAG}} = 0.49$) than for Butler pipelines ($\mu_{e_{Butler}} = 0.70$).

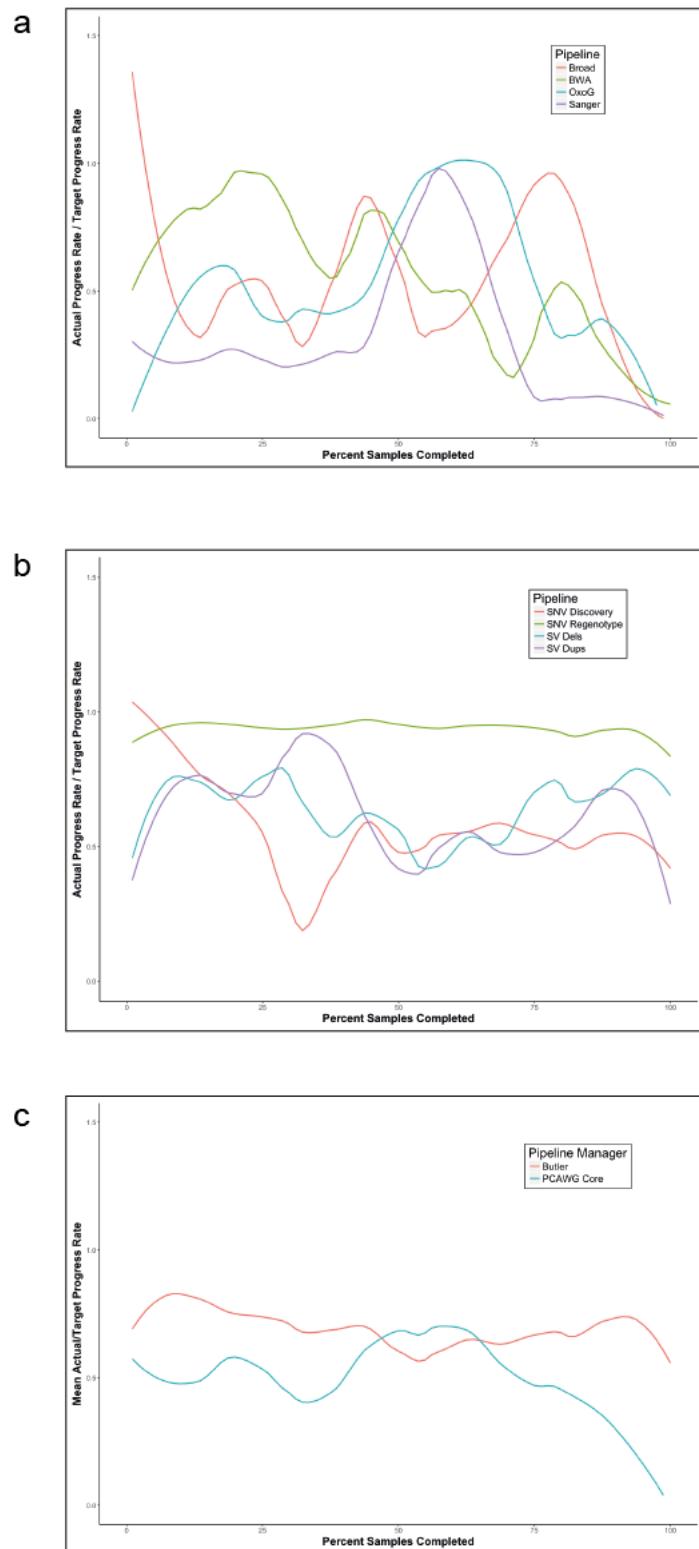


Figure 4.16: Effectiveness for "core PCAWG" and Butler-based pipelines. a - core PCAWG individual, b - Butler individual, c - mean "core" vs Butler.

Assessing the expected analysis duration for the two sets of pipelines we observed:

$$d_{PCAWG} = \frac{100}{\mu_{e_{PCAWG}}} = 2.04d_{opt} \quad (4.3)$$

$$d_{Butler} = \frac{100}{\mu_{e_{Butler}}} = 1.43d_{opt} \quad (4.4)$$

$$d_{PCAWG} = 1.43d_{Butler} \quad (4.5)$$

Thus, the estimated duration for PCAWG pipelines was 43% longer than that for Butler-based pipelines.

We further compared core PCAWG pipelines with Butler pipelines on the basis of uniformity of rate of progress through an analysis. Given a constant resource allocation an ideal analysis execution processes 1% of all samples in 1% of the analysis runtime. We divided the sample set into 100 equal size bins and measured the percentage of overall analysis time spent on processing each bin (Figure 4.17). Deviations from the diagonal indicate inefficiencies in data processing. Measuring this deviation we observed that PCAWG pipelines deviated 23.1% from the diagonal on average (min 0.0%, max 57.8%, sd 15.0%) while Butler pipelines only deviate 4.0% (min 0.0%, max 15.6%, sd 3.7%) from the diagonal on average, over the same sample set. This indicates that Butler pipelines are considerably less impacted by various causes that slow down an analysis (such as job and infrastructure failures).

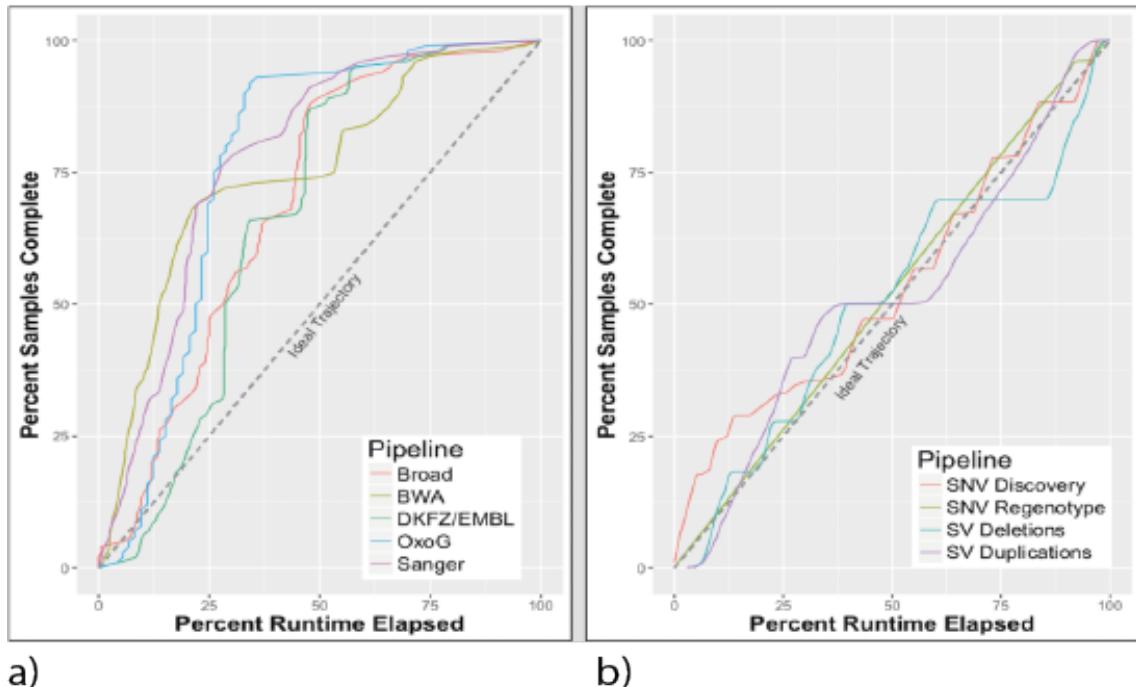


Figure 4.17: Butler vs "core PCAWG" analysis progress uniformity. a) - "core PCAWG", b) - Butler.

Carrying out large-scale scientific analyses in the cloud has its own distinct set of challenges from other types of scientific analyses. Throughout this work we have established four key areas of concerns that need to be met in order to facilitate the performance of such analyses by the end user - Provisioning, Configuration Management, Workflow, and Operations Management. We have built the Butler

framework to utilize existing robust open-source components where possible to fulfill the detailed requirements in the four areas thus described.

4.5 System Design Recap

Butler uses the Terraform provisioning tool in order to be able to create arbitrarily complex clusters in a cloud-agnostic manner, including artefacts such as Virtual Machines, networks, and security rules. We have used this capability to create test clusters on Amazon Web Services as well as rapidly creating and destroying production clusters on the EMBL/EBI's Embassy Cloud of over 180 VMs and associated network and security infrastructure, as necessary.

Butler uses the Saltstack framework to enable scalable and platform-agnostic capabilities including the installation and run-time configuration of software and servers. We have used these capabilities to develop configuration profiles for over 30 different software packages that are used within Butler, both to configure Butler itself as well as configuring the scientific software required by particular workflows. The packages configured and installed by Butler are as varied as - PostgreSQL Server, RabbitMQ, GlusterFS, Influxdb, Elasticsearch, dnsmasq, Collectd, Freebayes, Delly, Samtools, and others. The role-based configuration model that have been put in place allows the user to simply create a new Virtual Machine and give it appropriate roles, when the machine communicates with the Salt Master it will be configured fully to the state prescribed by its roles and able to carry out useful work within minutes.

Butler utilizes a scalable and robust Airflow framework for its workflows. Because Airflow workflows are Python programs the users have all the power and flexibility of Python and its extended libraries at their disposal. The fact that each workflow task is a separate entity that can run on any worker machine in the cluster allows Airflow to be extremely scalable.

To enable provenance tracking of scientific analyses we implemented a `tracker` module in Python that models the relationship between Workflows, Analyses, and Analysis Runs, the latter being the main execution unit of a workflow associated with a particular analysis and data sample. Using a PostgreSQL database we keep track of Analysis Run state transitions and execution history.

To further facilitate workflow configurability we implemented a hierarchical configuration mechanism using JSON-formatted configuration files that are specified at three levels of granularity and resolved into an *effective* configuration at runtime. The JSON configurations form part of the provenance trail for an analysis and are stored in a PostgreSQL database which has native support for this data type, including query language extensions[120].

Butler's Operations Management framework relies on two complementary systems - metrics, and logs. The metrics collection system is an agglomeration of tools that work together to harvest over 50 health metrics from each host and into a time-series database wherefrom a dashboarding engine presents the information in a series of dashboards. The log collection system similarly harvests application and server logs, filtering them down to extract useful information and storing it in an Elasticsearch index. Log information is then visualized in a set of separate dashboards. The two data collection and visualization systems provide the user with information at two granularity levels - the metrics system is more coarse-grained and gives a VM-level view of the health of the system, while the log system provides an application level

view with a finer grained resolution of the events that are occurring at any given time. Together, these two systems allow the user to have very clear visibility into the overall system health and detect any issues, whether they be individual machine or application crashes, or wider systemic events like network bottlenecks or outages.

Chapter 5

The Rheos Framework

In this chapter we describe a software framework called Rheos, which demonstrates an approach for reasoning about large genomic datasets utilizing concepts of service-orientation and data streaming in contrast with traditional genomic data analysis frameworks[54] that take a procedural batch-based approach. Our approach allows the users to make active tradeoff decisions between analysis time, cost, and quality as well as setting up precise operational Service Level Agreements, both between Rheos components, and between Rheos and external systems, as we describe below. We begin by describing a general framework of stream-based services and identify broad classes of such services based on their roles and capabilities. We use the general Rheos framework to implement several genomic data analysis use cases, namely germline SNP and Deletion variant calling. We scalably deploy Rheos onto an academic cloud computing environment running OpenStack and set up detailed monitoring to be able ensure operational control of the software at runtime. We conclude by testing Rheos on a real data set prepared by NIST’s Genome In a Bottle Consortium and comparing the results with the results generated by several leading tools. We find that the initial Rheos implementation is quite comparable to other methods in accuracy (98% sensitivity, 99% specificity in SNP calling, 85% sensitivity and specificity in deletion calling), while allowing the user to actively control the tradeoff between time, cost, and quality, which isn’t readily available to other tools. This establishes Rheos as an attractive and promising approach for large scale genomic analysis of the future.

5.1 General Framework Design

As already discussed in Chapters 1 and 2, the general problem consists of collecting DNA samples from a population of individuals under study, sequencing these samples using Next Generation Sequencing techniques, identifying the mutations that are present, annotating their functional impact and utilizing the obtained data in a downstream data analysis with research or clinical decision-making goals. While there is a great variety of possible downstream analyses that may be performed depending on the individual goals of the analyst, there is a fairly well established set of steps for processing of the raw NGS data into a set of annotated variants, and it is these steps that we target with this work. The typical approach that is in widespread use today is to collect a batch of samples and then process each sample individually with a sequence of separate tools, that may be described via a higher-

level workflow construct (such as in Figure 2.26, or using a framework like Butler, as described in Chapters 3, 4). There are, however, a number of factors that leave room for improvement in this model.

Data arrives from the sequencer as a random ordered collection of reads. These reads are mapped to a coordinate system of the reference genome. Reads are then sorted by coordinate and individual variant callers traverse the data set in a coordinate-wise fashion to perform the analysis. See Chapter 2 for more details on this process. This approach implies that data needs to be mapped before it is sorted, and sorted before it is variant called, which means that a single sample (Which can be anywhere between 150-300 GB in size currently) is traversed completely at least 4-5 times before any meaningful results are obtained. Additionally this implies that decisions need to be made up front about how much data to sequence, and analyze which virtually fixes the cost, time, and quality of the analysis before any data is seen. If we are able to abandon this approach in favour of a dynamic online integration of new data as it becomes available we open up the opportunity to achieve greater efficiency (because we are not waiting for the entire data set to be processed in one stage before going on to the next), and enable the active tradeoff between cost, time, and quality based on the data that is being observed. Since we progressively elaborate our data set as new data becomes available, we can choose to cut off our observations early if we achieved a desired quality level, or observed an event of interest (thus reducing time and cost), or we can choose to end an analysis by a certain time point and accept the results that have been achieved up to that point.

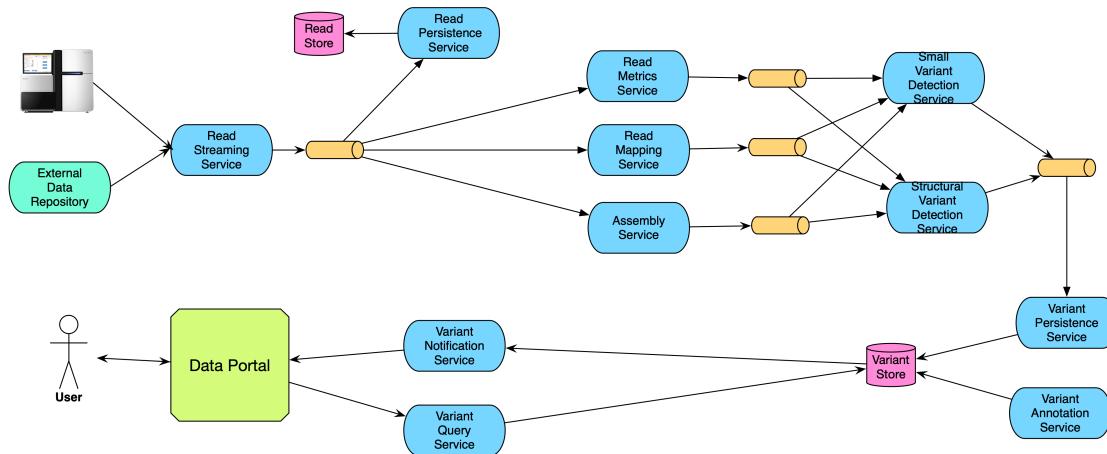


Figure 5.1: High level architecture of the Rheos framework.

This approach requires a new framework for genomic data processing as a continuous stream of data, because the current batch-based toolset is focused on processing entire files and doesn't allow real-time dynamic decision making about cost and accuracy. Additionally, a suite of new algorithms for online analysis is required, because core assumptions of currently existing algorithms are broken when data is not all available at the same time, and when it arrives in random order, rather than being coordinate-sorted. The general high level architecture of the Rheos framework the we have created to answer these challenges is depicted in Figure 5.1. Each blue bubble is a service - an encapsulated collection of data processing capabilities that operates on a stream of data and has well understood runtime characteristics that can be dynamically controlled. Each orange pipe is a distributed queue that stores

stream-based data in-flight and facilitates routing of information through the system. There are data stores established for data persistence where necessary, and a data portal to facilitate interaction with users.

The interface between existing file-based systems and the new stream-based Rheos system occurs at the Read Streaming service. This is where data is read from a traditional data source like a BAM file, and is transformed into a stream of reads. Useful metrics about the reads are collected by the Read Metrics Service. The reads are mapped to a reference genome using the Read Mapping Service and flow over to variant calling at the Small Variant and Structural Variant Detection services. Variants are persisted to a data store and are accessible to the user via the Data Portal both in pull model where the user can query specific variants, and in a push model where the user is notified about the identification of variants of interest. Each variant calling service maintains a set of models about the sample under analysis that are progressively elaborated as new data arrives. We measure the capacity of each service to perform its operations and increase and decrease resource allocation based on required throughput.

Since in this framework we are able to continuously integrate new data, measure outcome accuracy, and dynamically allocate resources based on desired operating characteristics, this establishes the overall set of capabilities that we set out to achieve. We dedicate the rest of this chapter to a detailed discussion of the Rheos framework architecture, design of individual services, their implementation, deployment, and experimental validation on a real data set.

5.2 Data Streaming Architecture

The overall technical architecture of the Rheos system is set up as a Service Oriented Architecture (SOA)[196] which is an information system architecture paradigm where the overall problem that the system is trying to solve is broken down into a collection of loosely-coupled components called services. Each service has a well defined interface of inputs that it accepts and outputs that it produces. Services can be combined and orchestrated together to produce the overall desired output for the system. A key distinguishing feature of this architectural approach is that each service can be individually optimized to fulfill its contract most efficiently helping break down some of the performance limitations brought about by the necessity to simultaneously tackle competing constraints in more monolithic information system designs. Additionally, within a services framework, the dependencies between separate services can be negotiated not only in terms of service interfaces, but also in terms Service Level Agreements which constitute Quality of Service promises made by one service to its dependents[99]. Because it is unlikely that a service designer will be able to accurately foresee all of the demands that will be placed on a service during its lifetime the SLAs provide a valuable feedback framework through which the service can be evaluated as it operates in production, as well as serving as a basis for negotiating evolving requirements between dependent services.

While general web services can support any data processing paradigm, in the Rheos framework we adopt a data streaming approach[18]. In this approach we assume that the input to any service is a randomly ordered sequence of messages $M = m_1, m_2, \dots$ where each message represents a fact about the underlying domain that the service reasons over, as well as some metadata, including an identifier, and a variety of

timestamps of interest. The content of each message may provide a datum, such as the measurement of a quantity of interest, or signal that a particular event has taken place. It is in general assumed that the data stream is infinite in size, that messages may arrive out of order, and that any message that is placed in the stream is observed at most once, and may, in fact, never be observed. Messages are typically not sent directly from one service to another, instead the transfer of messages is mediated by a queuing system using a publish-subscribe[62] model. Under this model each queue acts as a *topic*. Message producers can publish data to the topic, and message consumers subscribe to receive messages from the topic. A message is consumed from the queue only after all of the subscribed consumers have seen it. End users retrieve information from the system via a set of User Interfaces that support both push (notifications) and pull (querying) models of data retrieval. A more detailed description of the architectural aspects of the system follows:

5.2.1 Service-Oriented Data Streaming Model

A data stream $M_{s,d} = m_1, m_2, \dots$ is a sequence of datagrams transmitted over the network with the following properties:

- The stream has a source s and a destination d .
- A message m in the stream is a tuple of the form $(header, payload)$, where:
 - $header$ is a tuple of the form (id, \dots) that holds at minimum a unique identifier id for messages, and may hold additional metadata.
 - $payload$ is an arbitrary data structure that holds the informational content of the message.
- $|M| = \infty$ by assumption.
- Messages may not arrive at destination d in the same order that they were sent from source s .
- If $t_{i,s}$ is the time message m_i leaves the source s and $t_{i,d}$ is the time of arrival at destination d , then $\sup_i\{t_{i,d} - t_{i,s}\} = \infty$, i.e. a given sent message may never arrive at its destination.

A service $S = \{o_i\}$ is a collection of operations o_i that act on one or more input data streams $\{M_i\}$ to produce one or more transformed output data streams $\{M_j\}$. Specifically:

An operation O is a tuple of the form:

$$O = (i, o, p, f) \quad (5.1)$$

where:

- $i = \{M_j, j \in [0, K]\}$ is a set of $K \geq 0$ input data streams.
- $o = \{M'_j, j \in [0, L]\}$ is a set of $L \geq 0$ output data streams.
- $f : M^K \mapsto M'^L$ is a transformation function that produces messages m' in the output streams based on messages m observed in the input streams.

- $p = \{p_i\}$ is a set of potentially optional query parameters.

There are several distinct categories of operations that a service can perform on a set of input streams. We describe these here:

Windowing Function - Service S observes a sliding window, which is a sample of size n of messages from stream M_i and computes a summary statistic (see Figure 5.2) over the sample which is meant to be an approximation of the corresponding population parameter.

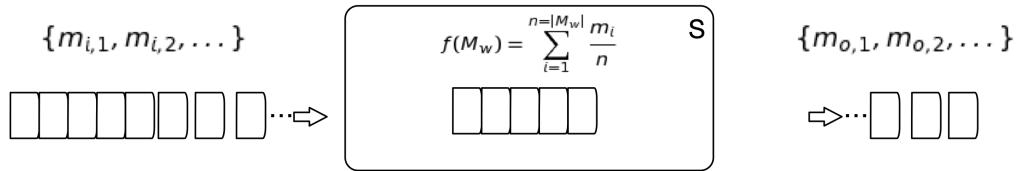


Figure 5.2: Service S computes a summary statistic over a window of messages from stream M

Decorator Function - Service S observes messages m_i and applies a function that augments (decorates) each message with additional attributes (see Figure 5.3) producing augmented messages m_o as output.

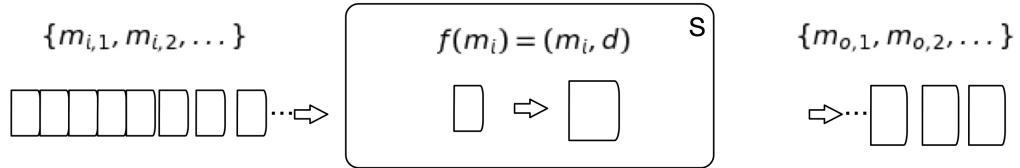


Figure 5.3: Service S augments messages from M with an additional set of attributes.

Filter Function - Service S observes messages m_i and applies a function $f : M \mapsto \{\text{True}, \text{False}\}$ that evaluates to a boolean value (see Figure 5.3). Only messages that map to True are emitted as output.

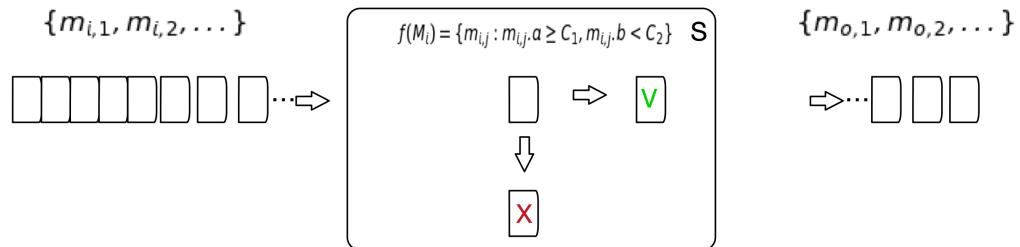


Figure 5.4: Service S filters messages from input stream M and only allows through those that pass the filtering condition.

Aggregator Function - Service S observes messages from N different streams $\{M_j : j \in [1, N]\}$ and applies a function $f : M_i^N \mapsto M_o$ that aggregates messages from these streams to produce its output (see Figure 5.5). Because aggregation happens over groups of messages that may not all arrive at the same time the service S requires a mechanism for keeping local state so that it can accumulate messages that have already arrived while waiting for those that are necessary to compute f yet have not been observed. The statefulness requirement of this type of service places an extra level of complexity (related to state-management and request routing) as well as inherent scalability limitations compared to stateless services[166].

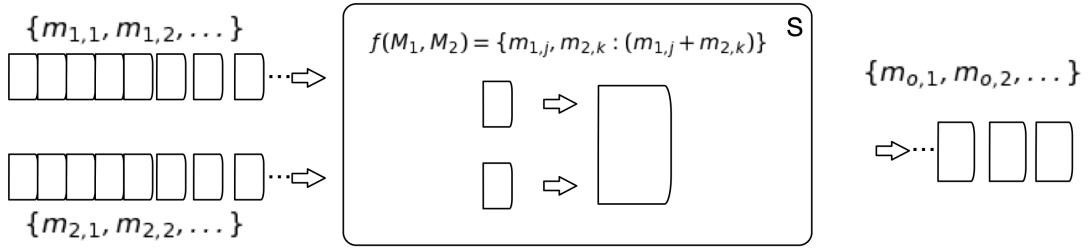


Figure 5.5: Service S integrates messages from multiple input streams M_i to produce an aggregated output stream M_o via f .

Local State Aggregator Function - Service S observes an input stream M_i which it integrates with a local (non-stream) queryable data store (see Figure 5.6). Messages m_i are integrated with query results q_i to produce an output stream M_o . This type of service also requires management of state and scalability concerns similar to the Aggregator service, especially when the local data store is itself distributed.

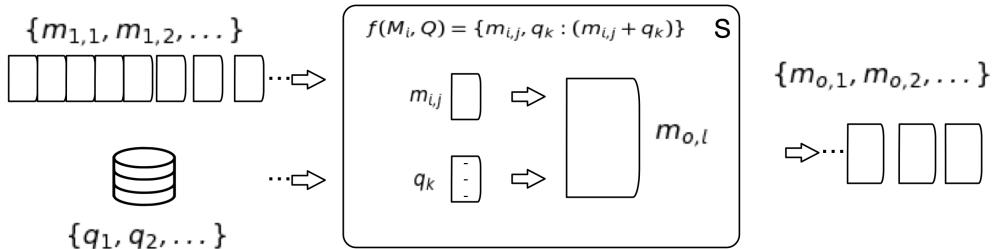


Figure 5.6: Service S aggregates m_i with query results q_i obtained from a local data store.

Persistence Function - Service S observes messages m_i and is responsible for persisting them to a data store where their contents can later be queried (see Figure 5.7). Although persistence of data to, and subsequent querying of data from, a store, such as a database, are comparatively more expensive operations than immediate reasoning over a live data stream, such mechanisms are necessary for situations where data may need to be accessed multiple times, or where data may need to be retained for audit purposes.

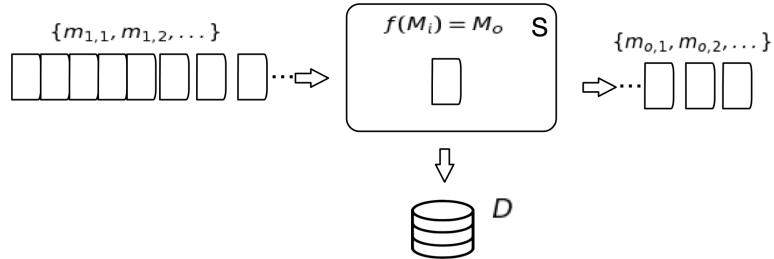


Figure 5.7: Service S processes messages m_i into persistent storage. The output stream M_o contains persistence confirmation and error events.

Query Function - Service S observes a stream of queries Q_i . The queries are fulfilled against a data store D and the results emitted via the output stream M_o .

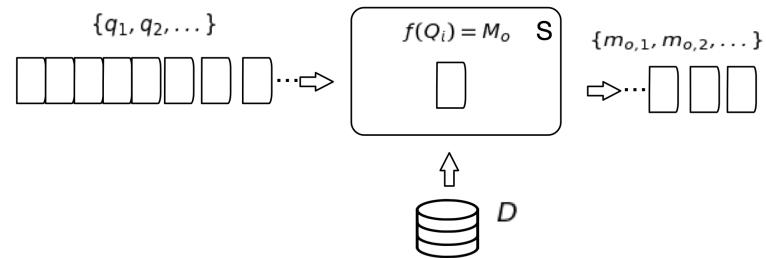


Figure 5.8: Service S filters messages from input stream M and only allows through those that pass the filtering condition.

The basic operations above can be combined to produce arbitrarily complex logic on data streams.

One of the key advantages of a service-oriented approach is that, because services are typically constantly executing, it naturally lends itself to an examination of the system's runtime characteristics. This applies to both service-internal characteristics that are related to each operation a service performs, as well as to external characteristics that relate to the contracts a service establishes with its dependencies. We consider both of these.

For each given operation $o_i \in S$ it is instrumental to understand the resource requirements of the operation on typical inputs and limiting factors that affect the efficiency with which the operation can be performed by S . Of particular interest are the per-operation profiles of:

- CPU utilization
- RAM
- Secondary storage
- Network utilization

If o_i is a long-running operation that takes multiple seconds to complete on average, a detailed distribution in time of each metric above may be necessary. If

the operation can be completed at a sub-second rate then summary statistics (min, max, mean, median, inter-quartile range, 90th, and 99th percentiles) may be sufficient. This level of understanding is necessary in order to make sure that the service can adequately deal with the incoming message stream while the messages are first loaded into memory, since subsequent retrieval from secondary storage is several orders of magnitude slower and may cause further delays in processing. If o_i is stateless, i.e. it does not require the storage and retrieval of any local state that depends on the content of each arriving message $m_j \in M_i$, then the service S can be scaled "horizontally"[217] with respect to o_i . Given that the performance-limiting condition of o_i is known (CPU, memory, etc.), the ability of S to efficiently deal with fluctuations in the rate of incoming messages M_i can be successfully achieved simply by adding and removing servers that execute S (see Figure 5.9), which can be done automatically[143]. If o_i is stateful and requires access to databases, or predictable request routing via sessions, then horizontal scalability may not be possible and thus, detailed understanding of the performance profile and performance-limiting conditions of o_i is even more important as vertical scaling of services is more expensive and challenging to accomplish, and may increase system complexity by necessitating data partitioning, for example[217].

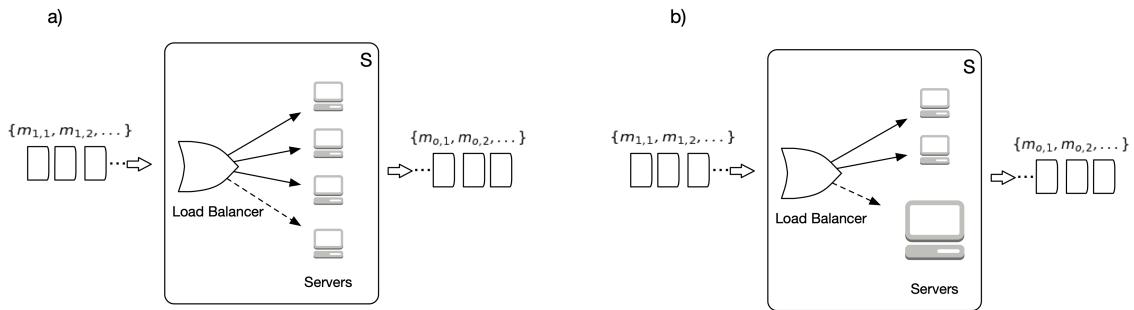


Figure 5.9: a) In horizontal scaling new servers are added and removed (dashed arrow) behind a load balancer as the rate of data stream M_i fluctuates. b) In vertical scaling more powerful servers need to be launched (dashed arrow) to replace smaller servers (with potential service outage) when the rate of M_i increases beyond capacity.

Assume service S implements operation o supported by n physical servers $V = \{v_j : j \in [1, n]\}$ by consuming a stream of incoming messages M_i . For a suitable time increment t , let $r_{M_i} = |M_i|/t$ be the incoming message arrival rate, and $r_{M_{o,j}} = |M_{o,j}|/t$ be the processing rate for server v_j . If $r_{M_i} > \sum_{j=1}^n r_{M_{o,j}}$, then S will not be able to adequately process all of the incoming messages from M_i and messages will either be lost or need to be backlogged while more servers are added to S to deal with the incoming message rate. Since commissioning new servers takes considerable time and the timing and magnitude of increases in r_{M_i} may be unpredictable, serious information loss may result if measures are not put in place to mitigate the message rate fluctuations.

A queue is the mechanism that we put in place to address this concern (see Figure 5.10). A queue Q is a message buffering system which consists of a set of n "topics" $P = \{p_i : i \in [1, n]\}$, where each topic is a tuple of the form $p_i = (D_p, B_p, C_p)$. Here $D_p = \{d_i : i \in [1, k]\}$ is a set of k data producers that put messages into Q , B_p

is a message buffer of max capacity N_{max} dictated by underlying server hardware characteristics, containing a sequence of messages $\{m_t, m_{t-1}, m_{t-2}, \dots, m_1\}$ that are accessible in a Fist In First Out (FIFO) manner, and $C_p = \{c_i : i \in [1, l]\}$ is a set of l consumers that are interested in observing messages from p_i .

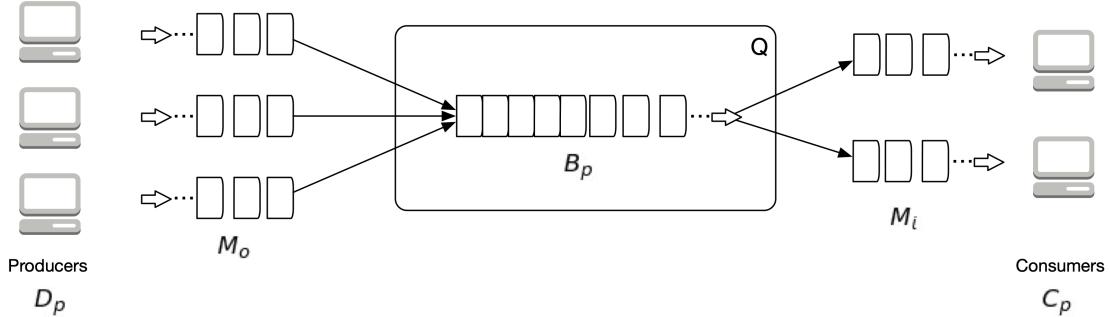


Figure 5.10: A queue Q establishes a message buffer B_p between a set of message producers D_p and consumers C_p , for a given topic p .

Messages arrive into a particular topic of Q from all producers D_p , and are marked safe for deletion only when all of the subscribed consumers C_p have observed a particular message. Thus, for topic p_i , the incoming message rate is $R_i = \sum_{j=1}^k r_{M_{o,j}}$ i.e. the sum of the message processing rates for all of the producers for this topic. The queue message processing rate $R_o = \min_{i \in [1, l]} \{r_{M_{o,i}}\}$ is the slowest message processing rate among all consumers. Assuming $R_i > R_o$ and that there are N messages presently in B_p , there remains $t = \frac{N_{max}-N}{R_i-R_o}$ time before queue overflow occurs. The situation should then be remedied by allocating additional hardware to Q or those services S_i whose consumers are slowest, until the condition $R_o \geq R_i$ can be reliably maintained. If the queue does reach its maximum capacity overflow measures need to be put in place. Depending on the data stream in question data loss may or may not be acceptable. If data loss is acceptable then overflow messages can be simply discarded. If data loss is not acceptable then producers must block waiting for additional queue capacity to become available. This not only degrades performance locally, but can have a drastic effect on the entire system if the effects are allowed to percolate through the complex distributed system. As message rates evolve through time with system load, the scheme above sets up a framework for flow control and hardware allocation within the architecture.

When designing a service-oriented system the interfaces of operations provided by the service are of utmost importance as they define the capabilities that the service offers to its clients. Of secondary, but also significant, importance is the set of Service-Level Agreements (SLAs)[232] that a service advertises. These SLAs are a set of commitments that a service makes to its clients that describe the operational characteristics of the service, such as:

Availability - Guarantees related to the service uptime, maintenance outages, disaster recovery, etc.

Throughput - The number of requests serviced per unit time.

Latency - The delay between a request being sent and a response being received.

Abandonment Rate - Proportion of requests that are never answered.

Error Rate - Proportion of well-formed requests that result in an error.

Based on the SLAs that are advertised by a given service, the services that depend on it can make assumptions about expected runtime behavior, and take action when expectations are not met. Furthermore, when requirements evolve and features are added to or removed from a service, the impact on the advertised SLAs helps communicate the full effect of the changes. Lastly, the costs of operating a service are more clearly understood through the SLA framework, where improvements to a particular SLA metric, such as Transactions-Per-Minute (TPM) can be transparently traced to a corresponding increase in operational costs.

The set of services $\{S\}$ that communicate over data streams $\{M_{s,d}\}$, mediated by a set of queues $\{Q\}$ with a set of established SLAs $\{L_s\}$ together form the overall framework of Rheos that is used to tackle the challenges of large-scale genomic data processing in a manner that enables active tradeoffs between the competing constraints of cost, time, and accuracy.

5.3 Domain-specific Problems

Having laid out the general data-streaming service-oriented architecture of Rheos in the previous section we now turn to a discussion of the set of actual domain-specific problems that need to be solved within the data-streaming paradigm in order to enable the comprehensive genomic characterization of large cohorts of samples within Rheos, as we have set out to do. We make use of the flow of data types from the most raw to the most refined (see Figure 5.11) to illustrate the challenges that need to be solved during transformation of the input data between each successive stage, first in summary form, and then in full detail, below.

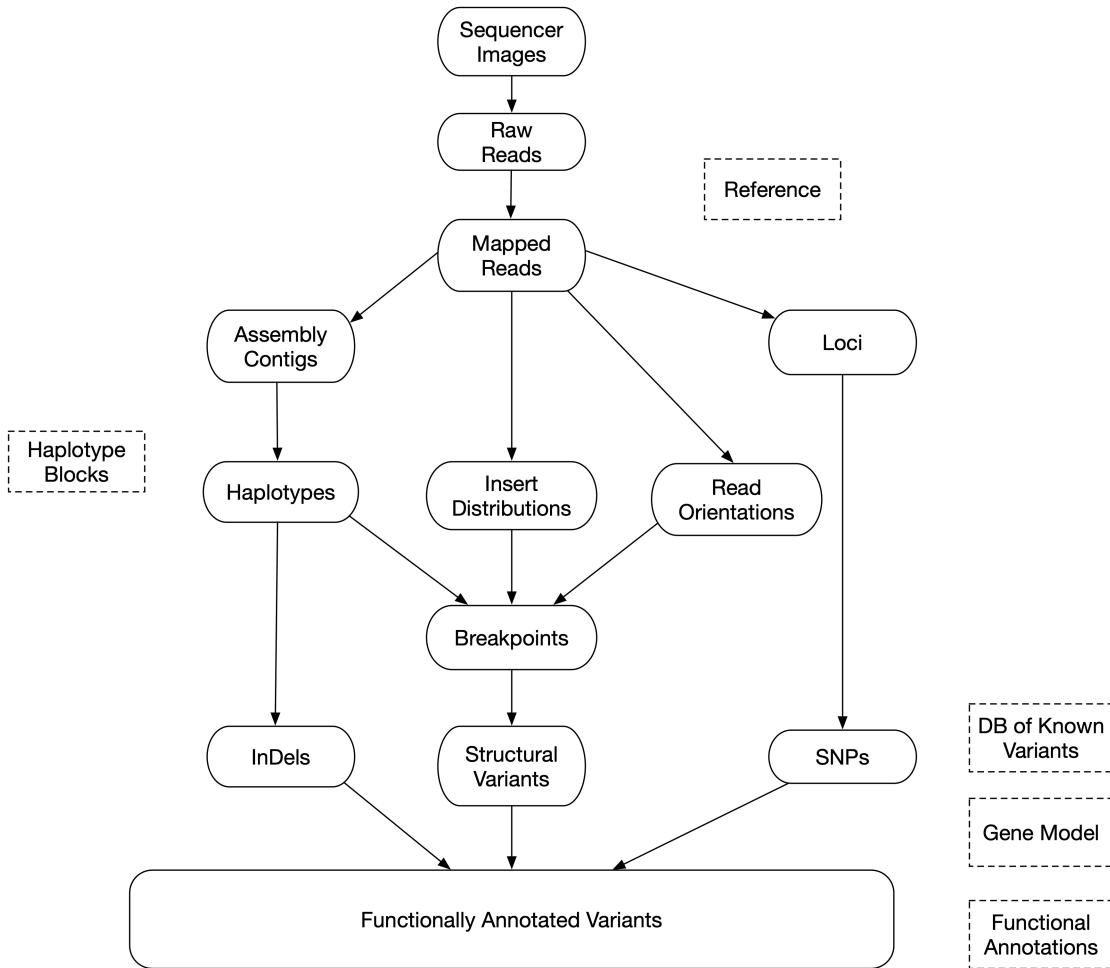


Figure 5.11: The conceptual flow of data types within Rheos from the most raw - Sequencer Images, to the most refined - a set of Functionally Annotated Variants.

The most raw data type that is produced from a sequencing experiment is the set of raw image files generated by the sequencer. Although, conceptually, processing of the raw images could also be accomplished within Rheos, it is presently outside of the scope of this work. Instead, we assume the most basic data type to be raw sequencing reads, as found in a FASTQ[34] file.

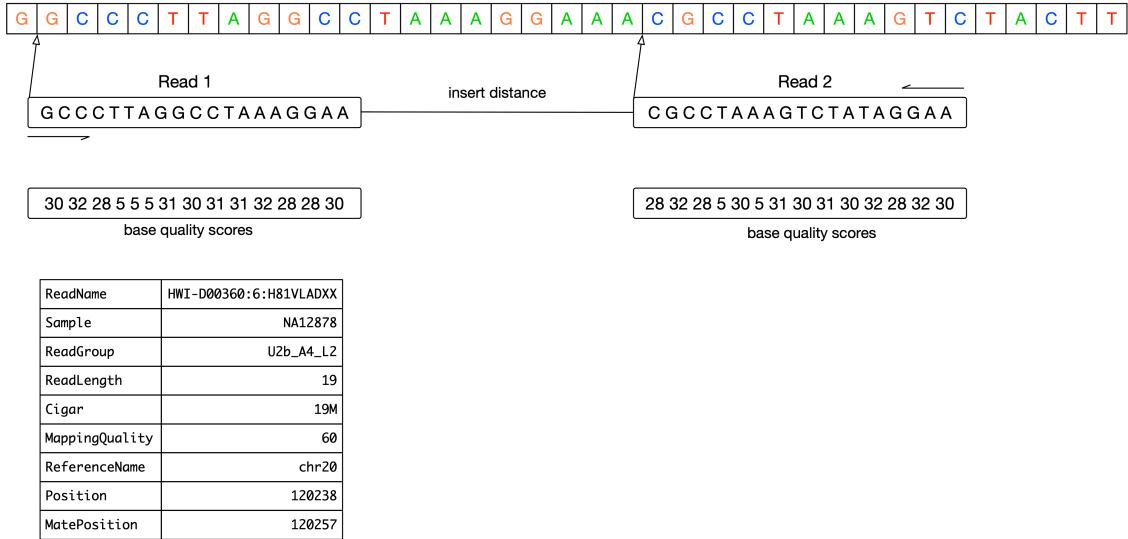


Figure 5.12: A read-pair that is aligned to the reference.

Each read is a tuple of the form:

$$r = (s_id, r_id, b, q, f_p) \quad (5.2)$$

where:

- s_id - is the sample ID, which is unique among all samples.
- r_id - is the read ID, which is unique among all reads for that sample.
- $b = \{b_1, b_2, \dots, b_n\}$ - is the sequence of DNA bases, where $b_i \in \{A, C, G, T, N\}$.
- $q = \{q_1, q_2, \dots, q_n\}$ - is the set of PHRED-scaled base quality scores corresponding to the probability that the base has been called incorrectly. See discussion on FASTQ format in Section 2.2.1 for details.
- $f_p \in \{True, False\}$ - is a boolean flag indicating whether this read is the first read in a pair.

5.3.1 Read QC Metrics

Section 2.2.3 of the Background chapter discusses various metrics of interest that are based on observations of read data and the tools that are used to collect them. Here we describe how to collect the most typical metrics in the streaming paradigm of Rheos. As before, there are per-read metrics such as Base Quality Distribution, and Adapter Sequence Presence, as well as per-sample metrics such as Average GC Content, Insert Distribution, Read Length Distribution, and others. The utility of these metrics is to be able to set up filters for low quality data as well as input for downstream variant-calling models (see Section 2.2.5 for example).

Assume that we are observing a stream $M_{raw} = \{m_i : m_i = (\text{header}, \text{payload})\}$ of read messages where the payload is a read r as defined above. Under the assumptions of Section 5.2 we know that the number of elements in the stream is unbounded. We are able to straightforwardly calculate incremental estimates for metrics such as

mean, variance, max, and min, but require more sophisticated structures for computing estimates of rank statistics such as median and other quantiles to maintain operations in bounded space. We use the following update rules for min, max, mean, and variance[229] calculations:

$$\min_k(M) = \begin{cases} m_k, & \text{if } m_k < \min_{k-1}(M) \\ \min_{k-1}(M), & \text{otherwise} \end{cases} \quad (5.3)$$

$$\max_k(M) = \begin{cases} m_k, & \text{if } m_k > \max_{k-1}(M) \\ \max_{k-1}(M), & \text{otherwise} \end{cases} \quad (5.4)$$

$$\mu_k(M) = \mu_{k-1}(M) + \frac{m_k - \mu_{k-1}}{k} \quad (5.5)$$

$$\sigma_k^2(M) = \frac{\sigma_{k-1}^2(M) + (m_k - \mu_{k-1})(m_k - \mu_k)}{k-1} \quad (5.6)$$

In order to set up the mechanisms to answer quantile queries the following definitions are used[77]:

- Given a set S of size n , and a quantile $\phi \in [0, 1]$, return $v \in S$ whose rank in sorted S is ϕn .
- An ϵ -approximate ϕ -quantile is a value v whose rank $r*(v) \in [n(\phi-\epsilon), n(\phi+\epsilon)]$.
- A quantile summary is $Q = q_1, q_2, \dots, q_l : q_1 \leq q_2 \leq \dots \leq q_l, q_i \in S, i \in [1, l]$ where each q_i has rank at least $r\min_Q(q_i)$ and at most $r\max_Q(q_i)$ in S , and $r\max_Q(q_1) \leq \epsilon|S|$, and $r\min_Q(q_l) \geq (1 - \epsilon)|S|$.
- A quantile summary $Q(\epsilon)$ is ϵ -approximate if it can be used to answer any quantile query with ϵ -accuracy.

We use two approaches for computing quantile summaries, one due to Greenwald and Khanna[83] is able to compute the quantile summary using $O(\log(\epsilon n)/\epsilon)$ space, and the other by Shrivastava et al.[198] computes the quantile summary in $O(\log(M)/\epsilon)$ when the values are integers in range $[1, M]$. Both algorithms work for a scenario where one node sees all of the data in a stream, but can also be generalized to topologies where the stream is observed by multiple nodes in parallel.

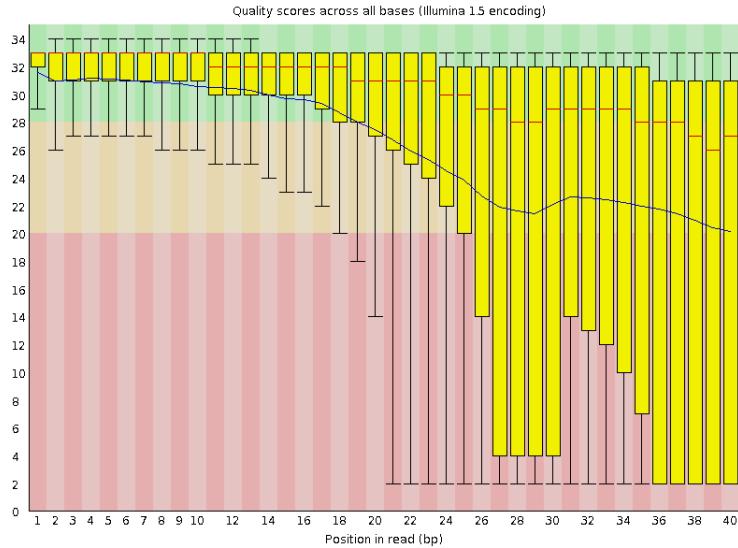
We provide several examples of QC queries of interest that are specified on a data stream:

Average Base Quality - As an assessment of the individual quality of each read we are interested in the average base quality so that we can filter out reads that are of low quality as a whole. We use a Decorator Function construct from Section 5.2.1.

Base Quality Distribution - The distribution of base quality scores per base position of a read and per sample are of interest to investigate the presence of systemic biases in base quality scores as a function of the position within the read.

Table 5.1: Definition of q_{av} which computes average base quality for a read

Inputs	$M_{raw} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = r = (s_id, r_id, b, q, f_p)$ as in 5.2.
Operation	$q_{av} = \frac{\sum_{i \in [1, r]} r.q_i}{ r }$
Outputs	$M_{out} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = r = (s_id, r_id, b, q, f_p, q_{av})$

Figure 5.13: Distribution of base qualities per read position, from <https://www.bioinformatics.babraham.ac.uk/projects/fastqc>. Large quality drop-off can be seen towards the end of the read.

Because PHRED-scaled quality scores are integers that fall in a fixed range $q \in [0, 96]$ building quantile summaries using the q-gram[198] approach is the most space-efficient. Because base quality scores need to be aggregated over many reads and tracked for many samples, a service that implements this functionality needs to keep local state, and the operation to update the quantile summaries based on incoming reads follows the Local State Aggregator pattern from Section 5.2.1.

Table 5.2: Definition of $updateQuantileSummaries()$

Inputs	$M_{raw} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = r = (s_id, r_id, b, q, f_p)$ as in 5.2.
Operation	$updateQuantileSummaries(r)$
Outputs	$M_{out} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (s_id, Q_{bqd})$.

Since the local state required for storing the quantile summaries may not fit in memory and may need to be persisted to disk, updating the summaries may be

too expensive to do for every single read that is observed in a read input stream. Instead, reads may be buffered into a set of reservoirs, triggering an update of the quantile summaries when the reservoir is full. The contents of the reservoir would then be purged and an updated set of quantile summaries $Q_s, bqq = \{q_i : i \in [1, max_bases]\}$, where each q_i is a quantile summary corresponding to the Base Quality Distribution at a particular read position, issued to the output stream (see Algorithm 4).

Algorithm 4: Updating quantile summaries for Base Quality Distribution.

```

Function UPDATEBQDQUANTILESUMMARIES( $r$ ) begin
     $reservoir \leftarrow GETRESERVOIR(r.s\_id)$ 
     $reservoir.ADDNEWREAD(r)$ 
    if  $reservoir.isFull$  then
         $summaries \leftarrow GETQUANTILESUMMARIES(r.s\_id)$ 
        for  $read$  in  $reservoir$  do
            for  $index, read.q$  in  $read$  do
                 $UPDATEQGRAM(summaries[index], read.q)$  // per [198]
        end for
         $PURGERESERVOIR(reservoir)$ 
         $OUTPUTQUANTILESUMMARY(r.s\_id, summaries)$ 
    end if
end function

```

Insert Size Distribution - The insert size distribution is an important metric because it is not only indicative of the overall quality of a sample's data, but it is also used by structural variant calling to find read-pairs that map abnormally far apart (indicating a deletion), or abnormally close together (indicating an insertion).

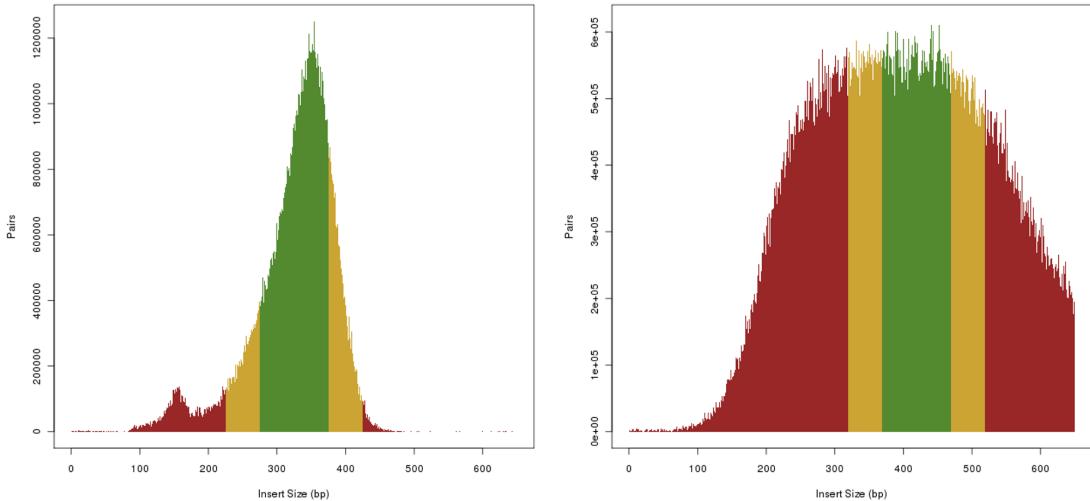


Figure 5.14: Distribution of insert sizes from two ICGC pancreatic cancer patients DO35138 and DO22154.[206]

Calculating this metric requires a stream of read-pairs, where both reads have been successfully mapped to the reference genome. Given a mapped read-pair (r_1, r_2) where each read has a beginning coordinate $r.pos$ and an end coordinate $r.end$, the insert size is $l = r_2.end - r_1.pos$. We are interested in the mean, variance and

quantiles of the insert size distribution. Because the insert length can be any size the quantile summary method of Greenwald and Khanna[83] is most appropriate for the quantiles. Read pairs are observed on the input stream and buffered in per-sample reservoirs. When a reservoir is full the read pairs are used to update and output an appropriate insert size distribution mean, variance, and quantile summary.

Table 5.3: Definition of *updateInsertSizeDistribution()*

Inputs	$M_{pair} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$, and $r = (s_id, r_id, b, q, f_p)$ as in 5.2.
Operation	<i>updateInsertSizeDistribution(s_id, r₁, r₂)</i>
Outputs	$M_{out} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (s_id, \mu_{isd}, \sigma_{isd}^2, Q_{isd})$.

Algorithm 5: Updating metrics for Insert Size Distribution.

```

Function UPDATEINSERTSIZE DISTRIBUTION(s_id, r1, r2) begin
    pairReservoir  $\leftarrow$  GETRESERVOIR(s_id)
    pairReservoir.ADDNEWREADPAIR(r1, r2)
    if pairReservoir.isFull then
        summary  $\leftarrow$  GETQUANTILESUMMARY(s_id)
        mu  $\leftarrow$  GETMU(s_id)
        sigmaSq  $\leftarrow$  GETSIGMASQ(s_id)
        for (r1, r2) in pairReservoir do
            insertSize  $\leftarrow$  r2.end - r1.pos
            UPDATEQUANTILESUMMARY(summary, insertSize) /* per [83] */
            newMu  $\leftarrow$  UPDATEMU(mu, insertSize) /* using Eq. 5.5 */
            newSigmaSq  $\leftarrow$  UPDATESIGMASQ(sigmaSq, mu, newMu, insertSize)
            /* using Eq. 5.6 */
        purgeReservoir(pairReservoir)
        OUTPUTINSERTSIZE DISTRIBUTION(s_id, newMu, newSigmaSq, summary)
    
```

Other QC metrics, such as those measuring GC Content Distribution, Read Length Distribution, etc. can be collected analogously.

5.3.2 Alignment

Section 2.2.2 of Chapter 2 provides an overview of the existing approaches in the extremely important and computationally intensive genome alignment stage of the overall NGS processing pipeline. In the overall data flow diagram (Figure 5.11), alignment is primarily responsible for transforming Raw Reads into Mapped Reads, but is also used in QC (insert size distribution, sample contamination) as well as in the construction and evaluation of local haplotypes for variant calling. In this section we describe the adaptation of already established read mapping best practices to the stream and services based domain of Rheos. Because of the generally independent

nature of individual read observations (except for read pairs), this read mapping problem is highly amenable to a stream-based approach. We begin by enumerating and describing the types of alignment tasks that the Rheos framework needs to be able to accomplish and follow up by describing how these tasks will be performed within Rheos.

Single read alignment to reference - Given a representation of the human reference genome we are interested in finding a coordinate relative to this reference where the given read best matches. If we are not able to find a high quality mapping the read should be flagged as unmapped.

Read pair alignment to reference - Given a pair of reads and an estimate of fragment size, attempt to find a high quality consistent mapping for both reads in a window around the expected fragment size.

Single read alignment to multiple references - Given a read and a database of several genome references from multiple species determine if the read has a high quality mapping in any of the references. This can be used for assessing sample contamination.

Candidate haplotype alignment to reference - Given a candidate haplotype i.e. a contiguous (potentially long) sequence locally assembled from a set of reads, align the sequence to the reference genome to identify locations of potential variation.

Single read split-alignment to reference - Given a read that does not align well to a contiguous region of the reference, search for an alignment where individual pieces of the read might align to separate and possibly distant locations on the reference, suggesting that the read spans a region of structural variation.

Single read alignment to reference Assume we are observing a data stream of raw unmapped short (<500 bp) reads $M_{raw} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = r = (s_id, r_id, b, q, f_p)$ as in 5.2. Using an existing human reference genome, such as *GRCh38*, we would like to align each read in $m.\text{payload}$ to produce two new output streams M_{aln} and M_{unaln} , depending on alignment results. M_{aln} , will contain messages from M_{raw} with additional attributes related to the alignment, as described in Section 2.2.1, replicating the information contained in a SAM alignment record (see Table 2.3). M_{unaln} shall contain reads from M_{raw} that failed to align to a reference sequence, with an additional flag *unmapped = true*.

For M_{aln} the following values are computed:

rname - name of the reference contig read is aligned to.

pos - 1-based position offset of the left end of the read alignment to the contig specified in qname.

mapq - Phred-scaled mapping quality representing probability that the read is misaligned.

cigar - CIGAR string (as in SAM specification Section 2.2.1).

flags - a tuple of flags as specified in the FLAG field of a SAM record.

Given that BWT and FM-index based algorithms[112],[123] have currently shown the best balanced performance characteristics on both simulated and real data (see Figures 2.16, 2.15) and the fact that these algorithms are amenable to straightforward parallelization, as in [113] for example, we adopt this approach as well in Rheos. Using this approach we will be able to locate k locations for potential matches of a seed d_i of read sequence $r.q$ in $\mathcal{O}(|r.q.d_i| + k)$ time using a data structure that takes $\mathcal{O}(|R|)$ space for a reference genome R . The seeds will then be extended using a version of Smith-Waterman[202] dynamic programming based alignment with affine gap penalties that can be performed in $\mathcal{O}(|R||r.q|)$ time and $\mathcal{O}(|r.q|)$ space based on [158] and [67]. The overall processing pipeline closely follows Figure 2.14 i.e. given a read r :

- Using Algorithm 3 find a set E of SMEMs of $r.q$ [123] using the FMD index formulation described in Section 2.2.2 in relation to BWA.
- Organize SMEMs e_i in E into co-linear chains of the form $C_i = \{(e_j, e_{j+1}, p)\}$, where e_j and e_{j+1} are neighbouring SMEMs in the chain and p is the mapping coordinate of e_j . Here co-linearity means that the SMEMs are in the same order and orientation on the query read and the reference genome, and a chain of maximal length is selected at each genomic location for the following processing step.
- For each chain, complete the read alignment between chain seeds using a vectorized SIMD-enabled implementation of Smith-Waterman local alignment, as in [67].
- Output alignment with the highest alignment score (based on number of mismatched bases and number of secondary alignments) if it's above a minimum quality threshold.

Because of the FMD index formulation used in finding seeds, both the read and its complement are considered at the same time. $rname$ and pos values are straightforwardly obtained from the coordinate and name of the leftmost position of the winning alignment. The *cigar* string (see Figure 5.15) is produced directly as a concatenation of the winning dynamic programming paths, and the SMEM seeds which are exact matches. $mapq$ converts alignment score to Phred-scale, and $flags$ encode a set of boolean values as per the SAM spec. This provides all of the information necessary for the output mapping.

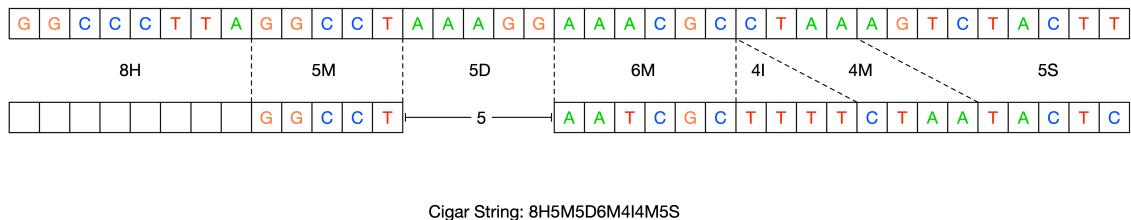


Figure 5.15: Example of an alignment CIGAR string.

If a co-linear set of seeds is not available, or the alignment score fails to reach the predetermined minimal score threshold then the read is emitted into the unmapped read stream with an appropriate flag set.

Table 5.4: Definition of $mapToReference()$

Inputs	$M_{raw} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = r = (s_id, r_id, b, q, f_p)$ as in 5.2.
Operation	$mapToReference(r, ref_id)$
Outputs	$M_{aln} = \{m_i : m_i = (\text{header}, r)\}$ where $r = (s_id, ref_id, r_id, b, q, f_p, rname, pos, mapq, cigar, flags)$
	$M_{unaln} = \{m_i : m_i = (\text{header}, r)\}$ where $r = (s_id, r_id, b, q, f_p, unmapped = true)$

The reference pointed to by ref_id when $mapToReference()$ is invoked consists of the typical data structures required by the FM Index i.e. the reference BWT, suffix array, occurrence array, as in [69], but encoding both the forward and reverse complement of the reference sequence as in [123] to produce the FMD index. Because the reference sequence is static, updated less frequently than once a year, the requisite data structures can be computed once offline and stored in secondary storage. During service initialization they are loaded in RAM and kept memory-resident for the duration of the operation of the service. Because, once computed, the reference index is read-only it can be accessed in a thread-safe manner by multiple concurrent threads without the need for explicit concurrency management. Because of the embarrassingly-parallel nature of read alignment this operation can be scaled up as necessary simply by adding servers, provided additional computational resources exist, and network capacity is not exhausted.

Read pair alignment to reference Since paired-end sequencing produces two reads that represent the opposite ends of a single molecule of approximately known size (known as insert size) it is possible to use knowledge about the insert size distribution along with corresponding pairs of reads to improve the quality of mapping for these reads, and even rescue mappings for reads that do not have a high quality unique mapping themselves but are anchored by a high quality mate. The mapping operation in this context proceeds similarly to the already described single read mapping but requires an input stream of read pairs along with a stream of sample QC metrics, including the empirical insert size distribution (as described in Section 5.3.1).

Depending on the result of the mapping operation for each read in a pair the output stream will contain pairs of reads of type

$$r_{aln} = (s_id, r_id, b, q, f_p, rname, pos, mapq, cigar, flags)$$

or

$$r_{unaln} = (s_id, r_id, b, q, f_p, unmapped = true).$$

In order to perform the $mapPairToReference()$ operation the service needs to have information about the insert size distribution for the sample the read pairs are originating from. It is notified with updated information about the insert size

Table 5.5: Definition of *mapPairToReference()*

Inputs	$M_{pair} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$, and $r_i = (s_id, r_id, b, q, f_p)$ as in 5.2.
	$M_{qc} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (s_id, \mu_{isd}, \sigma_{isd}^2, Q_{isd})$
Operations	$\text{mapPairToReference}(r_1, r_2, ref_id)$ $\text{updateInsertSizeDistribution}(s_id, \mu_{isd}, \sigma_{isd}^2)$
Outputs	$M_{aln} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$ and each read is of type r_{aln}
	$M_{unaln} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$ and each read is of type r_{aln} or r_{unaln} depending on the outcome of mapping.

distribution by the *updateInsertSizeDistribution()* operation which is subscribed to the appropriate message stream from the Read QC Service. This information is stored locally for each sample and may be cached. Assuming that the insert size distribution is approximately gaussian with mean μ_{isd} and variance σ_{isd}^2 , which is the case for well-behaved samples, a scoring metric can be constructed that favours paired alignments that fall close to the expected insert size. This metric helps select among non-uniquely mapping reads. For instance, BWA-MEM uses the following metric:

$$S_{ij} = S_i + S_j - \min -a \log_4 P(d_{ij}), U \quad (5.7)$$

Here S_i and S_j are the alignment scores for the individual reads in the pair obtained by single-end mapping via *mapToReference(r, ref_id)*. d_{ij} is the insert size implied by the mapping, $P(d_{ij})$ is the probability of observing an insert size larger than d under the assumption $D \sim \mathcal{N}(\mu_{isd}, \sigma_{isd}^2)$, a is a matching score, and U is a thresholding constant. Thus, given a set of possible mapping locations for each read in a pair, the joint mapping that maximizes the pairing metric is chosen.

mapPairToReference() outputs read pairs to two output streams. If both reads are mapped successfully then they are output to the M_{aln} stream. If at least one of two reads in a pair does not have a high quality mapping assigned through the paired mapping process, the pair is emitted through the M_{unaln} stream.

Pairing information is vital to downstream variant calling because it can both help with fragment assembly, when reads are properly paired and mapped, and signal the location of potential structural variants when reads are not properly paired and mapped within expected distances.

Single read alignment to multiple references When high average base quality reads fail to map to the reference genome this can be the result of sequencing errors, genomic rearrangements or sample contamination. It's important to be able to detect contamination both at the individual read level, to discard such reads from the analysis, lest they lead to spurious variant calls, as well as at the sample

level, where samples with an overabundance of contaminated reads may need to be discarded completely due to low confidence in the resulting call set. Contamination may occur in a variety of ways including sample-swap between tumour and normal DNA samples, between individuals, or accidental introduction of DNA from other species including other mammals, plants, bacteria, and viruses.

Because a relatively large collection of species' reference genomes has already been built up, one relatively simple way of detecting cross-species contamination is to classify reads by how well they align to a database of known references, if they fail to align to the human reference with a high enough quality. This can be accomplished with a general-purpose aligner like Bowtie or BWA, as well as with some purpose built approaches like Kraken[234] or Centrifuge[107]. While general purpose aligners appear to offer similar specificity and sensitivity to the purpose built tools, they are an order of magnitude slower and require approximately five times more RAM for storing the reference database[107]. Kraken breaks down the references into k-mers and uses k-mer counting to build a database of k-mer abundances and a taxonomic tree of species that can be searched for classification purposes, while Centrifuge uses the same FM index approach taken by Bowtie and BWA, but merges genomes from similar species into a graph structure and applies exact k-mer search when classifying reads from a sample (which is why it's faster than general purpose aligners since it avoids the expensive dynamic programming step).

Since the FM index implementation is necessitated by other alignment use cases it makes sense to adopt this approach for contamination analysis as well.

Table 5.6: Definition of $mapReadToReferenceDB()$

Inputs	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s_id, r_id, b, q, f_p, unmapped = true)$
Operations	$mapReadToReferenceDB(r)$
Outputs	$M_{aln} = \{m_i : m_i = (header, r)\}$ where $r = (s_id, ref_id, r_id, b, q, f_p, rname, pos, mapq, cigar, flags, contamination = true)$
	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s_id, r_id, b, q, f_p, unmapped = true, contamination = false)$

The key to performing the $mapReadToReferenceDB()$ is the construction of the reference DB itself. Assuming there are N genome references $G = \{g_i : i \in [1, N]\}$, create a joined reference $T = g_1\$g_2\$...\$_{N-1}g_N$ by concatenating all of the references interspersed with a set of sentinel characters $\{\$_i : i \in [1, N-1]\}$ that don't occur in G , are lexicographically smaller than any character in G , and having $\$_i < \$_j$ iff $i < j$. Record the index d of the occurrence of each $\$_i$ in T , such that $T[d_i] = \$_i$. Then construct and search the FM Index of T as usual. Given that a read r maps to some location l in T , $|\{d_i : d_i < l\}|$ identifies the rank of reference genome g_j in T that r belongs to, thus identifying the reference of origin. The alignment process itself may be tuned to only perform exact matching as in Centrifuge, or carry out dynamic programming alignment as well, depending on cost and performance requirements. Reads in the output stream that are successfully aligned to a reference that is part

of the database of contaminants are emitted with the corresponding reference ID and the contamination flag set.

Candidate haplotype alignment to reference Many of the modern leading variant callers[54, 78, 180] rely on local assembly of alternative haplotypes to detect variants with high accuracy. The haplotypes that are produced may be up to hundreds of kilobases long. These are aligned to the reference genome to reveal the location of potential variants that are implied by each alternative haplotype. The variants are subsequently evaluated in the context of read evidence to select the variants that are best supported by the data. This brings about the problem of aligning long sequences, potentially with many mismatches, to a reference genome. Additionally, long read technologies such as PacBio SMRT[179] and Oxford Nanopore Technologies[137] routinely generate reads that are many kilobases long and may enjoy increased use in sequencing projects due to their ability to resolve structural variation.

BWT and FM Index seed-and-extend approaches[122, 112] that have been successful for shorter read lengths have proven to be exceedingly slow or crash altogether for long reads, because of the extra backtracking required by the dynamic programming step in the presence of read sequence divergence from the reference, whereas a k-mer hash table chaining approach based on minimizers[181] was shown to be highly accurate and performant[126]. To take advantage of the improved alignment performance on long reads (30-70 times faster than BWA-MEM) we adopt Minimap’s approach when aligning locally assembled alternative haplotypes to the reference sequence.

Table 5.7: Definition of *mapLongReadToReference()*

Inputs	$M_{asm} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = r = (s_id, r_id, b, q, f_p)$.
Operation	$\text{mapLongReadToReference}(r, \text{ref_id})$
Outputs	$M_{aln} = \{m_i : m_i = (\text{header}, r)\}$ where $r = (s_id, \text{ref_id}, r_id, b, q, f_p, \text{rname}, \text{pos}, \text{mapq}, \text{cigar}, \text{flags})$

As described in 2.2.2 and [126] the reference index in this case consists of a hashmap of minimizers (see Figure 2.17) of the reference genome where the key is the minimizer sequence and the value is a list of locations of that sequence. As with BWT based indexes, the structure is static over time, can be pre-generated, stored on disk, and loaded into memory during operation. When a read arrives on the input stream M_{asm} the *mapLongReadToReference()* operation breaks the read into a set of minimizers and searches these against the reference hash table. Sets of co-linear exact matches are formed into chains. Sequences that are between matches in a chain are locally aligned with dynamic programming.

Single read split-alignment to reference When a high average base quality read fails to align to the reference with high mapping quality one of the reasons may be that the read covers a DNA double-strand breakpoint in the sample, at the

site of a structural variant that is longer than the size of the read. Such a read, if it were to be aligned to the sequence of the sample would map normally, but when mapped to the reference sequence, one part of the read maps to one location in the genome, and another part maps to a different location (see Figure 5.16). Such reads are called split-reads and they form an important source of signal for precise detection of structural variant breakpoints[177, 117].

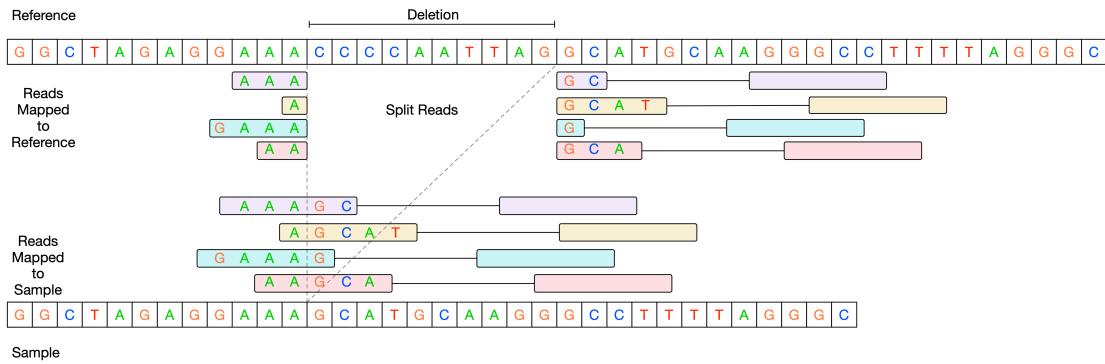


Figure 5.16: Split-reads at the site of a deletion breakpoint.

Although the techniques used for single read mapping, described above, can be used for split-read alignment there are several issues to consider that make the creation of a separate *mapSplitReadToReference()* operation desirable.

Since a split-read consists essentially of two or more separate pieces of varying lengths that may map to distant locations in the genome and have different orientations, determining where to "split" the read affects the quality of the subsequent mapping, as improper splits will generate many mismatches. *mapToReference()* is intended to be optimized for speed on well-behaved reads which constitute most of the data that is seen for a given sample, but as a result make a number of assumptions, such as a single k-mer size and co-linearity of matches, that are invalid or sub-optimal for split-read alignment. Furthermore, the data representation for an aligned read in *mapToReference()* of a single sequence b of bases, q of base qualities, and *cigar* string representing the alignment is not well suited to representing an alignment consisting of multiple, possibly distant, pieces of possibly different orientations, as this is not readily representable in CIGAR.

In current software that relies on writing SAM files as a medium for alignments, split-reads are represented as multiple records with the same ID, where one record is arbitrarily chosen to be primary (or representative) and others are marked supplementary (via an 0x800 FLAG value). Each record represents a separate piece of the split read where the unmatched portion is masked out via soft-clipping in the CIGAR string or is hard-clipped (see Section 2.2.1 for details on the SAM format). The details of the split-read alignment are captured in each read's optional SA tag in the form of a semi-colon separated list of strings that describe the supplementary alignments. In Figure 5.17, for example, there is a split read pictured that was obtained from a sample on chromosome 20 of the NA12878 individual in the 1000 Genomes Project, aligned by BWA-MEM, where one portion of the read aligns on the positive(+) strand, while another portion of the read aligns on the negative(-) strand some distance away. The yellow area in the figure describes the first part of the read which is labeled primary. The read is 250 basepairs long and the CIGAR

string is 68S81M101S, indicating 68 soft-clipped bases, followed by an 81 basepair match to the reference, and followed by another 101 soft-clipped bases. The 101 basepair segment has low base quality, so it likely consists primarily of sequencing errors, but the 81 basepair segment has base qualities around 30, which is relatively high. The SA tag reads - SA:Z:20,42589216,-,179S71M,60,0;, indicating that there is a secondary alignment mapped to chromosome 20, starting at position 42589216, on the negative strand, with CIGAR string 179S71M (179 soft-clipped bases followed by a 71 basepair match to the reference). This is indeed the case and the corresponding read is pictured in Figure 5.17 shaded purple. Investigating the SAM record for this mapping reveals that the actual record has CIGAR string 179H71M, i.e. the bases at the beginning are actually hard-clipped, not soft-clipped (soft-clipped bases are excluded from the alignment but the sequence is still preserved in the file record, whereas hard-clipped bases are excluded from alignment and the sequence is not recorded in the file). Such a split-read might imply the presence of deletion proximal to an inversion in the sample with respect to the reference sequence, although the evidence from the read should be investigated in light of the evidence from all of the other reads that overlap this locus to make an accurate determination of the presence of a potential variant in that location.

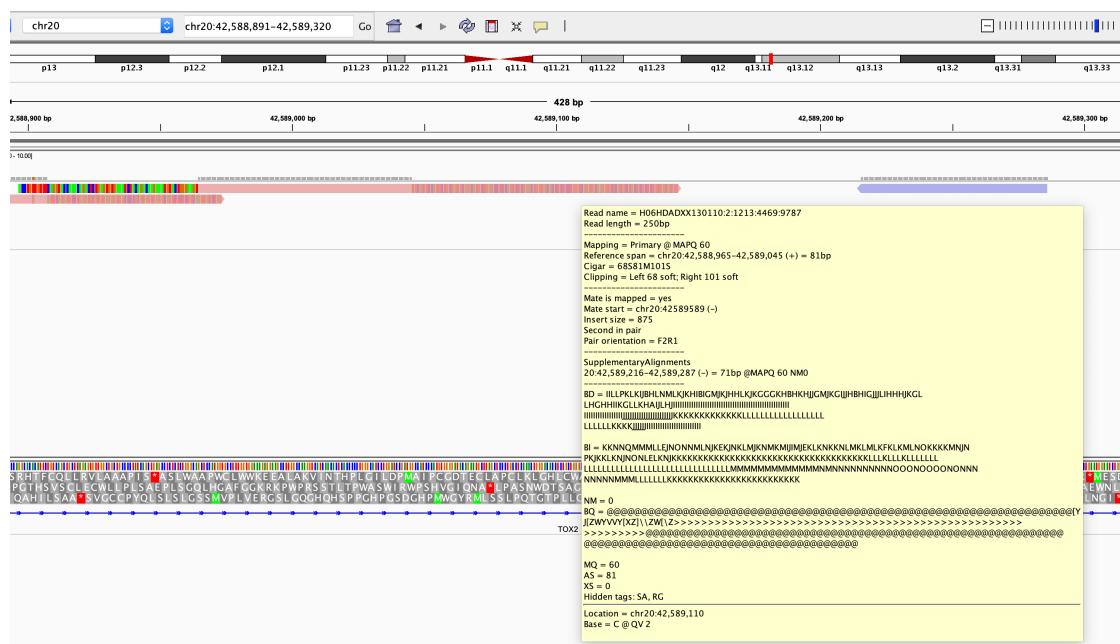


Figure 5.17: IGV viewer rendering of a split-read on chromosome 20 from NA12878 sample.

There are several issues with this scheme in addition to the apparent disagreement between duplicated information in the two records. The duplication is wasteful of space, and prone to errors as demonstrated above. There is no apparent ordering of the segments of the read that map to different locations because the next or previous segment are not specified for a given record, instead all secondary records point back to the primary. There is no well defined schema for the optional SA tag other than specifying a record separator. Thus, formats and contents of the encoding may vary between tools and even from record to record, without the ability to validate correctness of the encoding. Additionally, because the records in the SAM file are

typically either in random order or coordinate-sorted order, the records for a split-read are not easily locatable and in the worst-case the entire file may need to be scanned in order to get the full split-read alignment for a single read.

As Rheos does not rely on traditional genomics file formats for encoding, we chose a split-read representation that alleviates the issues above and encodes the entirety of a split-read alignment in a single record. Define:

$$r_{split} = (s_id, ref_id, r_id, b, q, f_p, aln) \quad (5.8)$$

Here $s_id, ref_id, r_id, b, q, f_p$ are as in Equation 5.2 defining a raw read. $aln = (a_1, a_2, \dots, a_n)$ - an ordered list of alignment segments, where each $a_i = (i, rname, pos, offset, len, mapq, cigar, strand, flags)$, and:

i - ordinal number of a_i in aln .

rname - name of reference contig to which a_i maps.

pos - position on $rname$ where a_i maps.

offset - offset on r_{split} where a_i begins.

len - length of a_i .

mapq - mapping quality.

cigar - CIGAR of the alignment.

strand - alignment strand, 0 for positive, 1 for negative. Strand disagreement between r_{split} and a_i indicates sequence inversion.

flags - various flags as in SAM format.

Table 5.8: Definition of $mapSplitReadToReference()$

Inputs	$M_{unaln} = \{m_i : m_i = (header, r)\} \text{ where } r = (s_id, r_id, b, q, f_p, \text{unmapped} = \text{true})$.
Operation	$mapSplitReadToReference(r, ref_id)$
Outputs	$M_{split} = \{m_i : m_i = (header, r)\} \text{ where } r \text{ is as defined in Equation 5.8.}$
	$M_{unaln} = \{m_i : m_i = (header, r)\} \text{ where } r = (s_id, r_id, b, q, f_p, \text{unmapped} = \text{true}, \text{split_align} = \text{false})$

The split-read alignment operates on reads that emerge unmapped from the regular alignment stage via $mapToReference()$. Alignment proceeds using the same general FM Index based framework as $mapToReference()$ but here each read is broken down into a series of progressively smaller k-mers. Each k-mer is aligned individually, and the alignment that maximizes aggregate alignment score across all k-mer sizes and candidate alignments is chosen.

The alignment operations that have been considered in this section present a set of distinct challenges on the basis of the gamut of query sequence and reference database size combinations. Each scenario presented admits optimization with respect to its specific parameters. Existing aligners typically attempt to solve all or most of these problems with a single approach, and thus suffer from inability to fully optimize each specific use case. Our treatment, that separates the different alignment scenarios into distinct operations, with well delineated optimization criteria, and even into potentially separate services that may run on separate physical machines, allows us to fully optimize each operation without negatively impacting others.

5.3.3 Simple Germline SNP Calling

SNPs are the most abundant and widely studied type of genetic variant[37]. Section 2.2.4 provides an overview of existing methods for germline SNP calling. For a human diploid sample when calling variants on one of the autosomes (chromosomes 1-22), germline SNP calling comes down to selecting between three alternative models for each genomic locus based on the set of reads that are observed - homozygous reference, heterozygous, or homozygous variant (see Figure 5.18).

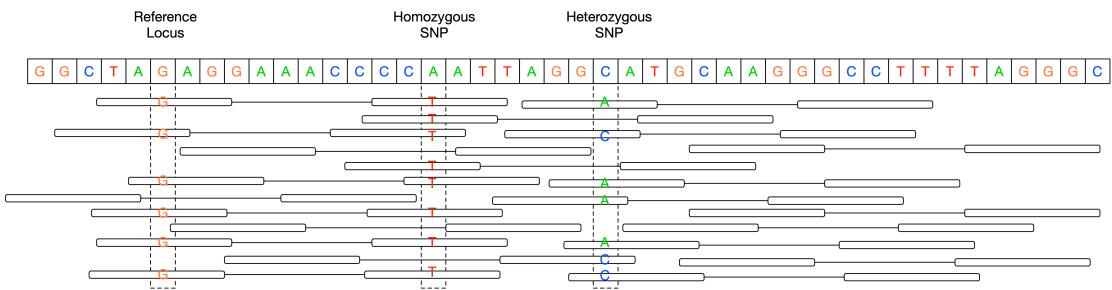


Figure 5.18: Examples of genomic loci that match the reference sequence, have a heterozygous SNP, and a homozygous SNP.

Although the most accurate currently available variant callers model the region around a SNP via local assembly of haplotypes[54, 180, 78], a simpler approach assumes that every locus is independent and can still achieve high accuracy. For instance, samtools[121], which makes the independence assumption, has recently been compared to several local assembly based methods in [134] and was shown to have a slightly higher false-negative rate than platypus, freebayes, and Haplotype Caller, but also a slightly lower false-positive (FPPM = false positives per million bases) rate (see Figure 2.25). Additionally, in [180] Table 1, samtools has the highest sensitivity, but also the highest false discovery rate (FDR) on SNPs. Thus, the simpler independent locus model should be sufficient for calling germline SNPs and is of interest in the context of the data streaming approach of Rheos.

The random order in which reads appear in a Rheos data stream means that a new approach is required to successfully implement germline SNP calling on this data. Specifically, all of the existing tools for variant calling assume that all of the reads for a given sample have been observed, mapped to the reference, sorted by increasing reference coordinate, and stored in a SAM or BAM file. Variant calling (see Section 2.2.4) proceeds by traversing the data in a coordinate-ordered fashion, loading all

of the reads that overlap a given locus into memory, into a structure called a read pileup (see Figure 5.19), and evaluating a set of alternative models for each locus, typically in a Bayesian framework, selecting either the maximum-likelihood estimate (MLE) or the maximum a posteriori (MAP) estimate as the called genotype.



Figure 5.19: A read pileup over a locus with a heterozygous SNP.

The Bayesian framework that is typically adopted follows this form (see also Section 2.2.4):

$$P(G|D) = \frac{P(G)P(D|G)}{\sum_i P(G_i)P(D|G_i)} \quad (5.9)$$

Here some prior distribution $P(G)$ of genotypes at a locus is assumed. This can be an uninformative prior (for instance one that assigns equal probability to all possible genotypes), or it can be a prior based on a population genetics model, or empirically obtained distribution of genotypes in a given population. The genotype likelihood $P(D|G)$ under assumptions of independent observations (reads) and independent errors, factors into a product of individual observation likelihoods i.e. $P(D|G) = \prod_j P(D_j|G)$. Each observation $P(D_j|G)$ is further subject to a sequencing error probability ϵ_j , derived from the recorded base quality for the read over the locus being considered. Thus, $P(D_j|G) = 1 - \epsilon_j$ when $D_j = G$ and $P(D_j|G) = \epsilon_j$, otherwise. Genotype likelihoods are then computed as in Equations 2.7 for samtools, and similarly Equations 2.13, 2.18, 2.15 for freebayes, Equation 2.20 for GATK, and Equation 2.21 for platypus. A site is called variant if the probability of having at least one non-reference allele at the locus exceeds the probability of having 0 non-reference alleles. A genotype is assigned if the ratio of the genotype with the highest likelihood to the genotype with the second highest likelihood exceeds some pre-determined constant, i.e., assuming genotype $g \in [0, 1, 2]$ denotes the number of reference alleles at the locus, and, for example, given $\mathcal{L}(g_2) > \mathcal{L}(g_1) > \mathcal{L}(g_0)$, assign the genotype $g = 2$ if $\frac{\mathcal{L}(g_2)}{\mathcal{L}(g_1)} > c$, where $c > 1$ is some user-set threshold. The calling

proceeds locus-by-locus in coordinate-ordered fashion, and the results are written to a file in VCF format (see 2.2.1) where each row represents a variant.

The process above is efficient in that each locus is looked at only once in order to be able to make a call, but it is inefficient in that all of the data needs to have been observed, mapped, and sorted in order to even begin calling. The streaming approach that we adopt in Rheos allows us to begin calling immediately as we start seeing data, in return for needing to re-evaluate a locus multiple times as new data arrives. Because each read is immediately integrated into the model once it is observed, the state of our model of the genome is at all times consistent with all of the read evidence that has been observed to date. Each locus then simply provides an estimate of the underlying genotype based on the data that is available at that time. Initially, these estimates may be inaccurate, but accuracy improves as more and more data arrives and the estimates are refined in an iterative fashion.

We are thus interested in an iterative update rule that, given a current set of genotype likelihood estimates at a locus, that incorporate evidence from n reads that have already been observed, when the read $n+1$ arrives, is able to produce an updated set of genotype likelihoods that is consistent with evidence from all $n+1$ reads, as well as any other auxiliary structures necessary for producing a variant call record. To do this we adopt a sequential Bayesian framework, where the posterior distribution of genotype likelihoods at step n becomes the prior (albeit unnormalized) for step $n+1$.

Let:

Symbol	Description
R	The reference sequence.
S	The sample sequence.
$D = \{d_i : i \in [1, n]\}$	The set of n observations seen to date.
d_i	i 'th read observation.
g_i	Genotype of sample S at locus i . $g \in [0, 1, 2]$ counts the number of reference alleles at locus i
$P_n(G_i)$	Probability of genotype $G = g$ at locus i , after seeing n observations d that overlap locus i .
$\mathcal{L}(d_n G_i)$	Genotype likelihood function at locus i for the n 'th observation.

We take $P_0(G_i)$ to be the prior probability of G at i . After observing the first data point d_1 which has base quality (probability of error) ϵ_1 , we calculate:

$$\begin{aligned}\mathcal{L}(d_1|G = 0) &= 1 - \epsilon_1 \\ \mathcal{L}(d_1|G = 1) &= \frac{1}{2} \\ \mathcal{L}(d_1|G = 2) &= \epsilon_1\end{aligned}$$

if d_1 is different from reference, and:

$$\begin{aligned}\mathcal{L}(d_1|G=0) &= \epsilon_1 \\ \mathcal{L}(d_1|G=1) &= \frac{1}{2} \\ \mathcal{L}(d_1|G=2) &= 1 - \epsilon_1\end{aligned}$$

if d_1 matches the reference. Consequently, by Bayes' Rule:

$$\begin{aligned}\mathcal{L}(G=0|d_1) &\propto \mathcal{L}(d_1|G=0) * P_0(G=0) \\ \mathcal{L}(G=1|d_1) &\propto \mathcal{L}(d_1|G=1) * P_0(G=1) \\ \mathcal{L}(G=2|d_1) &\propto \mathcal{L}(d_1|G=2) * P_0(G=2)\end{aligned}$$

Now, assuming that we have observed n data points and calculated $\mathcal{L}(G|D_n)$ (likelihood of genotype given n data points D_n). We obtain a new data point d_{n+1} and calculate:

$$\begin{aligned}\mathcal{L}(d_{n+1}|G=0) &= 1 - \epsilon_1 \\ \mathcal{L}(d_{n+1}|G=1) &= \frac{1}{2} \\ \mathcal{L}(d_{n+1}|G=2) &= \epsilon_1\end{aligned}$$

if d_{n+1} is different from reference, and:

$$\begin{aligned}\mathcal{L}(d_{n+1}|G=0) &= \epsilon_1 \\ \mathcal{L}(d_{n+1}|G=1) &= \frac{1}{2} \\ \mathcal{L}(d_{n+1}|G=2) &= 1 - \epsilon_1\end{aligned}$$

if d_{n+1} matches the reference. Using this likelihood and the posterior from step n as the prior for step $n + 1$, again by Bayes' Rule, we obtain:

$$\begin{aligned}\mathcal{L}(G=0|D_{n+1}) &\propto \mathcal{L}(d_{n+1}|G=0) * \mathcal{L}(G=0|D_n) \\ \mathcal{L}(G=1|D_{n+1}) &\propto \mathcal{L}(d_{n+1}|G=1) * \mathcal{L}(G=1|D_n) \\ \mathcal{L}(G=2|D_{n+1}) &\propto \mathcal{L}(d_{n+1}|G=2) * \mathcal{L}(G=2|D_n)\end{aligned}\tag{5.10}$$

Equation 5.10 provides the general update rule for incorporating new evidence about a given locus i having already made and recorded n observations, upon making observation $n + 1$. This iterative approach is mathematically equivalent to the batch update approaches such as [121], that collect all of the data about a locus before evaluating genotype likelihoods. The posteriors obtained by equation 5.10 can be used directly to evaluate relative likelihoods of the possible genotypes, and to assign genotypes via a likelihood ratio, for example, but in order to obtain a proper posterior probability mass function over the genotypes they need to be normalized, to obtain:

$$P(G=g|D) = \frac{\mathcal{L}(G=g|D)}{\sum_{g' \in G} \mathcal{L}(G=g'|D)}\tag{5.11}$$

This equation can be used to obtain a probability estimate of any given genotype, having observed any given amount of data about a specific locus.

Armed with the update rule for integrating read evidence we are ready to define a data stream operation of Local State Aggregator type that is responsible for processing a stream of aligned read pairs, updating its local state with the genetic variation evidence contained in the reads, and emitting a stream of updated genomic loci (see Fig. 5.19) that can be used by downstream services for variant calling.

Table 5.9: Definition of *updateLociFromRead()*

Inputs	$M_{aln} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$ and each read is of type r_{aln} .
Operation	$\text{updateLociFromRead}(r, ref, current_{loci})$
Outputs	$M_{loci} = \{m_i : m_i = (\text{header}, L)\}$ where $L = \{l : l = (s_id, ref_id, contig, pos, alt, gl_0, gl_1, gl_2, dp, ro, qr, ao, qa)\}$

A Locus l for SNP calling represents a single genomic location for a single sample and a particular reference genome. For each Locus we keep track of the following fields:

s_id - Sample ID.

ref_id - Reference Genome ID.

contig - Name of the reference contig.

pos - Reference-relative position of the Locus on the contig.

alt - The alternative allele at this locus (if any alternative alleles have been observed).

gl_0 - Likelihood of homozygous alternative genotype at this locus, based on the data seen so far.

gl_1 - Likelihood of heterozygous genotype at this locus, based on the data seen so far.

gl_2 - Likelihood of homozygous reference genotype at this locus, based on the data seen so far.

dp - Read depth - number of reads observed at this locus so far.

ro - Number of reference observations at this locus.

qr - Sum of base qualities of reference observations at this locus.

ao - Number of alternative observations at this locus.

qa - Sum of base qualities of alternative observations at this locus.

The Locus Processing Service which is responsible for implementing *updateLociFromRead()* maintains a local data store of Locus objects, ideally in memory. When a new mapped read arrives via the input stream, the service walks along the read and for each position on the read compares it to the respective position on the reference sequence. It retrieves the Locus corresponding to this position and, depending on whether the read matches, or is different from the reference sequence, updates the Locus genotype likelihoods (and other fields) according to the update rule of Equation 5.10. Those Locus objects that have been updated, and have alternative allele observations (i.e. potential variants) are added to a list of objects that should be emitted by the service as output of *updateLociFromRead()*. Since a read alignment consists of not only matches and mismatches with respect to the reference sequence, but also encodes insertions and deletions as an array of CIGAR elements (see 2.2.2), and the alignment may occur on the forward or the reverse strand of DNA (requiring reverse complementing) special care needs to be taken during read processing to account for all of the possible cases of simultaneous read and reference traversal. Read processing proceeds as in the following algorithms:

Algorithm 6: Process read and reference in tandem to find matching CIGAR elements.

```
Function PROCESSREADPAIR(read_pair, ref) begin
    for read in read_pair do
        strand ← read.strand
        cigar_els ← read.cigar_els
        ref_offset ← 0
        read_idx ← read.query_start
            /* Alignment does not necessarily start at index 0. */

        for el in cigar_els do
            if el.type = MATCH or el.type = DEL then
                if el.type = MATCH then
                    updated_loci
                    ← HANDLEMATCH(read, read_idx, el.length, ref_offset, ref)
                ref_offset ← ref_offset + el.length * strand
                    /* both matches and deletions consume reference sequence.
                     Reads that align on the negative strand consume reference
                     backwards. */

            if el.type! = DEL and el.type! = SOFT_CLIP then
                read_idx = read_idx + el.length      /* All elements but deletions
                                               and soft-clips consume read sequence. */
```

Algorithm 6 is responsible for locating the genomic coordinates of those parts of the alignment that are flagged as matches (a match may still have differences from the reference). This is accomplished by traversing the CIGAR elements of the alignment and updating the coordinate offset of the read, the reference, or both, depending on the type of CIGAR element that is encountered. When a CIGAR element corresponding to a match is encountered it is processed by Algorithm 7, which is responsible for updating the set of loci that the matching region overlaps with the evidence for (or against) variation that exists in the read. In Algorithm

Algorithm 7: Process a portion of the read that maps to reference as a match, updating all loci that it overlaps.

```

Function HANDLEMATCH(read, read_idx, match_length, ref_offset, ref) begin
    seq  $\leftarrow$  read.seq[read_idx : read_idx + match_len]
    qual  $\leftarrow$  read.qual[read_idx : read_idx + match_len]
    if read.strand = 1 then
        ref_start  $\leftarrow$  read.position + ref_offset - 1
    else
        /* When read is on negative strand sequence is reverse complemented
        and matching starts from end of reference. */
        seq  $\leftarrow$  REVERSECOMPLEMENT(seq)
        ref_start  $\leftarrow$  read.end + ref_offset - match_length - 1
    read_ref  $\leftarrow$  ref[ref_start : ref_start + match_length]
    for pos_idx in seq do
        cur_ref  $\leftarrow$  ref_start + pos_idx
        cur_locus  $\leftarrow$  ALL_LOCI.GETLOCUS(cur_ref)
        p_error  $\leftarrow$  BASEQUALITYTOPROBABILITYOFEERROR(qual[pos_idx])
        cur_locus.dp  $\leftarrow$  cur_locus.dp + 1
        cur_locus.gl_het  $\leftarrow$  cur_locus.get_het * 0.5
        if seq[pos_idx] != ref[pos_idx] then
            cur_locus.alt  $\leftarrow$  seq[pos_idx]
            cur_locus.gl_ref  $\leftarrow$  cur_locus.gl_ref * p_error
            cur_locus.gl_hom  $\leftarrow$  cur_locus.gl_ref * (1 - p_error)
            cur_locus.ao  $\leftarrow$  cur_locus.ao + 1
            cur_locus.qa  $\leftarrow$  cur_locus.qa * qual[pos_idx]
        else
            cur_locus.gl_ref  $\leftarrow$  cur_locus.gl_ref * (1 - p_error)
            cur_locus.gl_hom  $\leftarrow$  cur_locus.gl_ref * p_error
            cur_locus.ao  $\leftarrow$  cur_locus.ao + 1
            cur_locus.qa  $\leftarrow$  cur_locus.qa * qual[pos_idx]

```

7, the DNA strand to which the read maps determines whether the read sequence needs to be reverse complemented and whether the match offsetting proceeds from the beginning or the end of the reference sequence. After offsets are computed the actual matching always proceeds moving forward (in the direction of increasing coordinates) along the reference sequence. At each location the reference and read sequences are compared and the underlying Locus model is updated based on the update rule of Equations 5.10. Those loci that have a non-zero count of alternative allele observations are added to a list of loci that potentially harbour variants and will be emitted from the Locus Processing Service as a stream M_{loci} for further downstream processing by the variant caller.

The Variant Calling Service - is responsible for actually making SNP variant calls based on a set of user-defined calling criteria, such as a region of interest, variant quality, genotype likelihood, number of supporting observations, etc. The service translates between a set of Locus models maintained by Rheos and an external data format (namely VCF). This service does not operate on a stream of data, but instead acts as a query service that can be invoked by the user ad hoc with specified parameters to answer queries of interest.

For each locus inside the queried region the Variant Calling Service evaluates the set

Table 5.10: Definition of *callVariants()*

Operation	<i>callVariants(region, calling_threshold)</i>
Outputs	<i>outputVCF</i>

of genotype likelihoods at that locus and calculates normalized genotype posterior probabilities using Equation 5.11. The variant quality is defined to be the Phred-scale transformed probability that a site is homozygous reference, i.e. sites that, based on the read evidence, have a very low probability of being reference, under the Phred-scale transformation end up with a high variant quality (see Equation 2.11). Variants that exceed *calling_threshold* will be emitted as records to the output VCF file. It is possible to envision more sophisticated versions of the Variant Calling Service that provide extended querying and filtering capabilities.

5.3.4 Germline Structural Variant Calling

As Rheos aims to provide comprehensive sample characterization it is not enough to only be able to call single nucleotide polymorphisms, the ability to call other variants, such as indels and structural variants is also required. Here we examine an approach for calling a specific class of structural variants, namely deletions, using a stream of mapped read pairs made available by the Read Mapping Service's *mapPairToReference()* operation.

In general, structural variant callers typically make use of three types of evidence for the presence of structural variation - read depth, discordantly mapped reads, and split-reads (see Section 2.2.5). Here we utilize discordantly mapped reads i.e. those read pairs that align farther away from each other than expected in order to assess the presence of sequence deletions in the sample with respect to the reference.

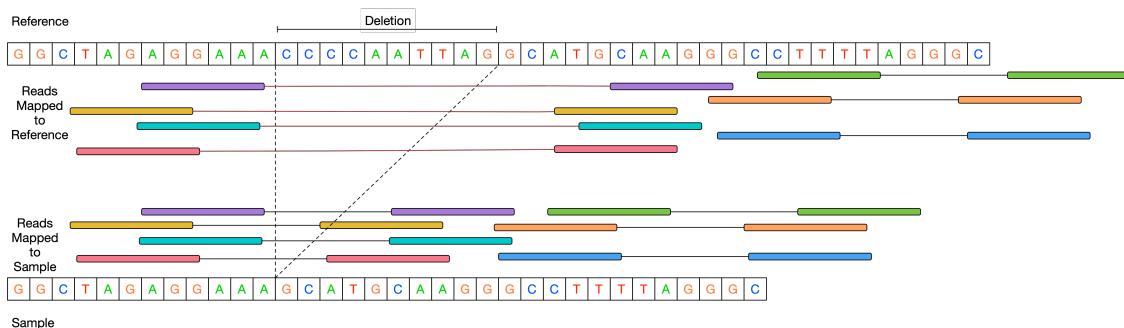


Figure 5.20: Effect on read-pair mapping distance for reads overlapping a deletion.

In Figure 5.20 the sample being sequenced has a deletion with respect to the genome reference, namely the sequence "CCAAATTAG" is deleted. When sequencing with read pairs, the term "insert size" refers to the size of a DNA fragment that is being sequenced, less the size of the sequence adapters on both ends (see 5.21). Thus, assuming that both reads in a pair have been mapped, and have mapped in the proper orientations (5' towards 3' on both strands). If the Read Mapping Service emits the read pair as (r_1, r_2) where r_i are as previously defined and $r_1.pos, r_1.end,$

$r_2.pos$, $r_2.end$, and assuming without loss of generality that r_1 is the read that maps on the positive strand, then the insert size $l_{r_1, r_2} = r_2.end - r_1.pos$.

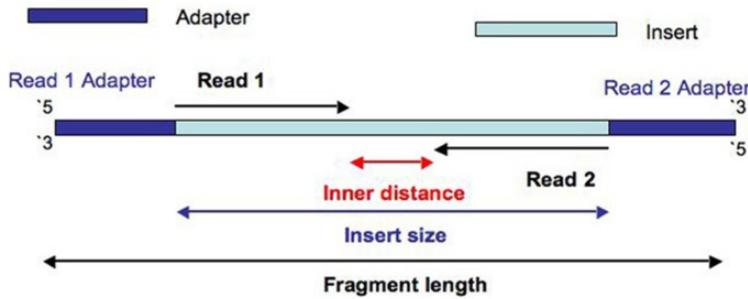


Figure 5.21: Read-pair insert size is the size of the DNA molecule being sequenced minus the length of adapters on both sides[214].

As can be seen from Figure 5.20, for those reads that span the site of the deletion the inferred insert size is larger than the actual insert size (approximately by the size of the deletion), since the read are mapped farther apart on the reference than they are on the sample. Such read pairs are said to be discordantly mapped. Since the insert size distribution for a given sample is relatively predictable (see Figure 5.14), it is possible to detect the presence of deletions in a sample by identifying clusters of read pairs that have an inferred insert size larger than some predefined threshold. Such an approach is already used by several structural variant callers[177, 117] and we adopt it in Rheos with several key innovations.

Detection proceeds in two stages performed by two different services. The Insert Size Filtering Service is responsible for listening to the general stream of read pairs and filtering out all of the pairs that don't meet the requisite criteria. The Insert Size Clustering Service is responsible for consuming a stream of discordantly mapped read pairs, clustering the reads that are located close to each other, and outputting those clusters that have sufficient evidence for being considered sites of deletions.

For the Insert Size Filtering Service we define the following operation:

Table 5.11: Definition of *filterDiscordantPairs()*

Inputs	$M_{aln} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$ and each read is of type r_{aln}
Operation	<code>filterDiscordantPairs($r_1, r_2, \text{min_mapq}, \text{mad_threshold}, \text{min_sample_size}$)</code>
Outputs	$M_{disc} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$ and each read is of type r_{aln}

Here, the *filterDiscordantPairs()* operation subscribes to the stream of mapped read pairs that is produced by the Read Mapping Service. The goal is to produce a stream of read pairs that are suitable for clustering. The following conditions must be satisfied in order for a pair to be considered suitable:

Both reads in a pair must be mapped - the Read Mapping Service actually emits some read pairs where only one of the reads is mapped into the mapped reads stream.

Both reads in must be mapped to the same contig - Read pairs where individual reads map to different contigs may be indicative of genomic translocations (another type of structural variant), but are not suitable for the detection of deletions, since different contigs are actually different physical molecules.

Both reads must have mapping quality $\geq \text{min_mapq}$ - A mapping quality of at least 30 (the default value) on the Phred scale, for instance, implies a probability of no more than 10^{-3} that a read is mapped to the wrong location on the reference.

Insert size must be $> \text{MAD} * \text{mad_threshold}$ - the Median Absolute Deviation is taken as a measure of the central tendency of the distribution of insert sizes that is robust against extreme outliers, which can be quite common. Typical fragment size of the DNA that is being sequenced is in the hundreds of nucleotides long, whereas certain inferred insert sizes can reach millions of bases due to structural variants or spurious mappings. Thus, it is desirable to have a metric similar to variance that will not be sensitive to these extreme values. Given a set of insert sizes $L = \{l_i : l_i = r_2.end - r_1.pos\}$, define $\text{MAD}(L) = \text{median}(|l_i - \text{median}(L)|)$. The cutoff of $\text{mad_threshold} * \text{MAD}$ places a lower bound on the size of deletions that can be detected with this method. A value of 5 is arbitrarily chosen as the default for mad_threshold .

Since obtaining an accurate estimate of median insert size and MAD requires observing a sample of read pairs, the Insert Size Filtering Service does not emit any reads at first, but accumulates observations in a local data store while collecting enough information to obtain reliable estimates of these metrics. min_sample_size (10^5 by default) read pairs are collected before filtering begins. Once the metrics are obtained, collected reads are filtered first and released, before processing any of the remaining reads on the stream.

Read pairs that satisfy all of the conditions above are emitted in an output stream for consumption by the Insert Size Clustering Service. The main points of control for the user are the min_mapq and mad_threshold parameters. Decreasing min_mapq increases sensitivity by allowing more reads through while simultaneously decreasing specificity. Likewise, decreasing mad_threshold increases sensitivity and allows detection of smaller size deletions, at the expense of allowing "less surprising" insert sizes through the filter. Choosing a higher min_sample_size provides more accurate estimates of median insert size and MAD at the expense of a higher delay before producing filtered output, and higher peak memory requirements for the Insert Size Filtering Service. Ideal parameter values may be project dependent.

The Insert Size Clustering Service is responsible for consuming the stream of discordantly mapped reads and figuring out locations of read clusters in genomic space that may be the site of potential deletions. Those read clusters that have sufficient evidence are then emitted as variant calls. Because discordantly mapping read pairs are fairly rare (out of 6705731 mapped read pairs on Chromosome 20 of sample NA12878 analyzed as part of this work, only 2663 or 0.04% successfully passed through the Insert Size Filtering Service with $\text{min_mapq} = 30$ and $\text{mad_threshold} = 5$), and the addition of a single new read pair can only potentially affect a small number of variants, it does not make sense to re-evaluate the clustering model for each new pair that comes in. Instead, the Insert Size Clustering Service accumulates discordantly

mapped read pairs from the input stream and will process them as part of an ad hoc query or a periodically scheduled execution, producing an updated call set as a result. We define the service’s operations in Table 5.12.

Table 5.12: Operations of Insert Size Clustering Service

Input	$M_{disc} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = (r_1, r_2)$, and $r_i = (s_id, r_id, b, q, f_p)$ as in 5.2.
Operations	$\text{addReadPair}(r_1, r_2)$ $\text{callDeletions}(\text{region}, \text{insert_size_threshold}, \text{min_read_support})$
Output	$M_{del} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = V = \{v_i : v_i = (s_id, \text{contig}, pos, length, vqual, R)\}$ is the set of called variants, and $R = \{r_i : r_i = (r_1, r_2)\}$ is the set of read pairs supporting the variant call.

Here $\text{addReadPair}()$ is an operation that reads the stream of discordantly mapped reads and stores each read pair in the service’s data store. The data store is an in-memory data structure that consists of an array for general read processing and an Interval Tree for performing queries for finding those reads that support a particular putative variant. The Interval Tree is a type of balanced binary search tree (described in [44], for example) that allows efficient querying of which intervals overlap a particular query interval. The tree can be constructed in $\mathcal{O}(n \log n)$ time, $\mathcal{O}(n)$ space, and answers queries in $\mathcal{O}(\log n + m)$ time, where n is the total number of intervals stored and m is the number of intervals being returned by the query (see Figure 5.22). Insertions into the tree after construction take $\mathcal{O}(\log n)$ time.

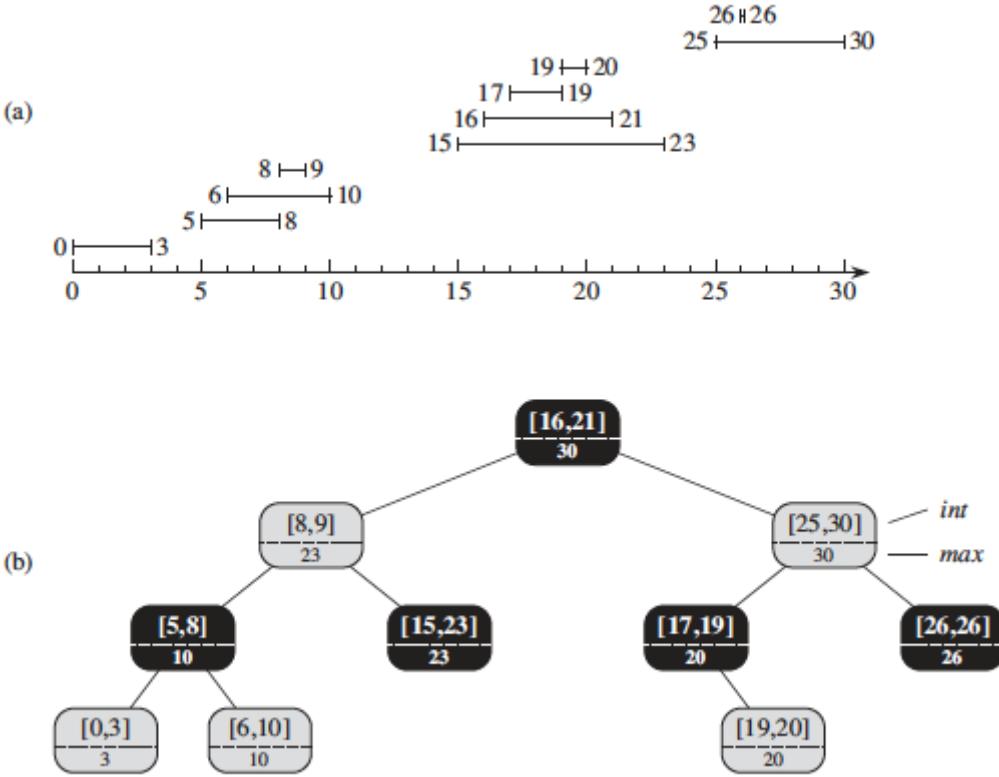


Figure 5.22: a) Intervals arranged on the number line. b) An Interval Tree constructed from the same intervals. Each node contains an interval and the maximum value of any interval in the subtree rooted at that node.[44].

Invocation of `callDeletions()` uses the data accumulated by `addReadPair()` to perform the actual deletion calling. The calling proceeds as follows:

1. For each read pair determine the center point of the interval spanned by the pair.
2. Using pair centers perform Kernel Density Estimation[185, 169] to estimate the distribution of the location of pair centers.
3. Using the estimated density detect regions with local maxima (clusters) that become locations for putative deletions.
4. Using the sites of putative deletions query the Interval Tree for a list of reads that overlaps each site.
5. For each site and set of read overlaps call the deletion boundary to be the maximal interval that is spanned by the inner distance (see Fig. 5.21) of all overlapping read pairs.
6. Based on the width of the deletion remove all overlapping reads whose inferred insert size is greater than `deletion_width + insert_size_threshold`.
7. Output a deletion if the number of supporting reads after filtering above is $> \text{min_read_support}$.

The key approach used in this method for the clustering of read pair interval centers is Kernel Density Estimation. This approach has been used as a highly successful and efficient one-dimensional clustering method in a variety of settings[89, 110, 45], but has not been applied in the context of genomic variant calling to date.

If we have a one-dimensional data set of iid observations (x_1, x_2, \dots, x_n) from some unknown distribution and we wish to estimate the pdf f we can do so using:

$$\hat{f}_h(x) = \frac{1}{n} \sum_{i=1}^n K_h(x - x_i) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x-x_i}{h}\right)$$

Here K is a non-negative real-valued integrable windowing function, and h is a smoothing parameter called bandwidth. A variety of kernels are possible. In this method we use the gaussian kernel as a way to attenuate the influence of each read pair center with distance from that center.

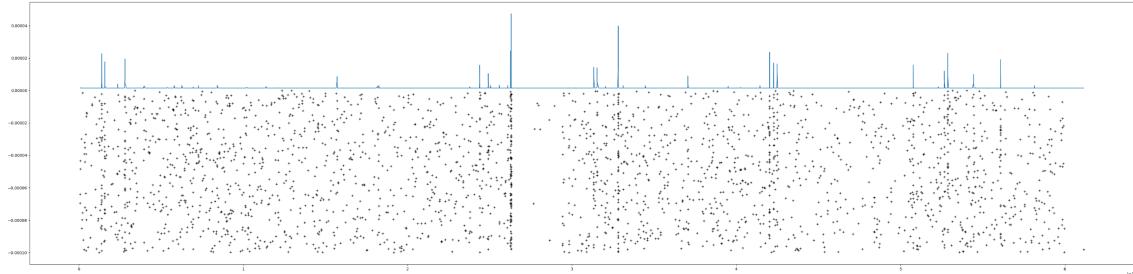


Figure 5.23: Centers of discordantly mapped read pairs on Chromosome 20 of NA12878 (plotted as points below x-axis) along with the KDE estimate plotted above x-axis.

In Figure 5.23 we demonstrate the results of applying KDE with a gaussian kernel and a bandwidth of 100 on the 2663 discordantly mapped read pairs from Chromosome 20 of the NA12878 sample, sequenced by the Genome In A Bottle[245] consortium and processed by Rheos. The x-axis specifies integer genomic coordinates from 0 to 6.3×10^7 and the y-axis is real-valued. Each point below the x-axis represents the coordinates of a centre for a single read-pair. The y-axis values for each point are randomly generated from a small interval in order to obtain visual point separation on the graph. The graph above the x-axis shows the estimated density function based on the 2663 points, with peaks indicating the position of clusters, and hence, the location of putative deleted regions. The location of peaks is obtained via a peak finding algorithm that returns the maximum value in an n-neighbourhood of points.

Once the location of the center points of putative deletions is known it is important to determine deletion size and evaluate the amount of support that exists for each deletion call. To achieve this `callDeletions()` makes use of the Interval Tree data structure that has been constructed from all of the read pairs. The Interval Tree is queried for all read pairs that overlap the position of a given deletion center. This query is answered in $\mathcal{O}(\log n + m)$ time.

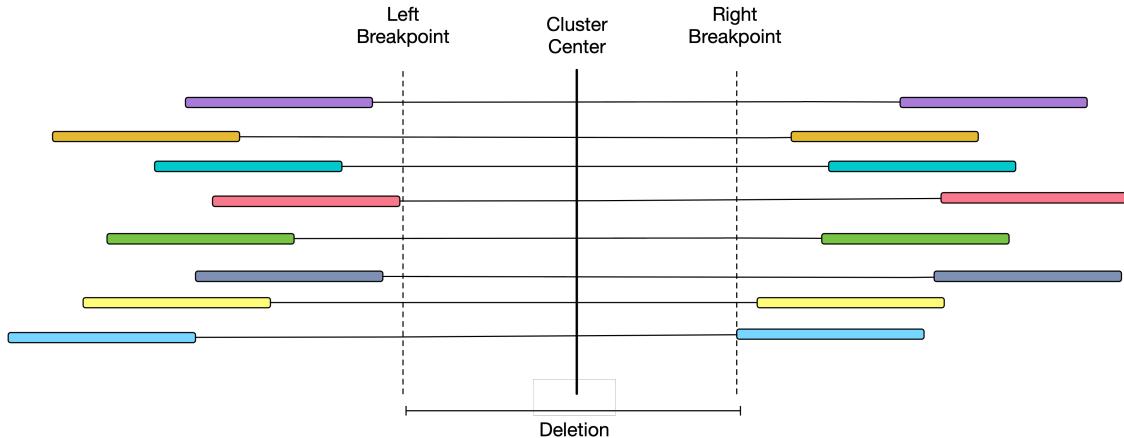


Figure 5.24: A cluster of reads spanning a germline deletion.

Given a set of read pairs $R = \{(r_1, r_2) : r = (r.\text{pos}, r.\text{end})\}$ spanning the site of the deletion we are interested in determining the extent (position, and size) of the deletion. We take the deletion site (see Figure 5.24) to be an interval $[\max_{r_1, i \in R} (r_{1,i}.\text{end}), \min_{r_2, i \in R} (r_{2,i}.\text{pos})]$, based on the following argument. If the deletion is homozygous, then no read in the sample should map into the deletion sequence, thus the deletion should be entirely contained in the inner distance between reads spanning the deletion. If the deletion is heterozygous, then the reads that are discordantly mapped come from the haplotype with the deleted allele, and should thus, also not have any sequence map into the deletion, again containing the deletion entirely in the inner distance between reads. Thus, the deletion boundaries are in the interval between the largest end coordinate for a read to the left of the deletion center, and the smallest start coordinate to the right of the deletion center. Given the location and width of a putative deletion it is necessary to determine whether the deletion has sufficient support in the reads to be considered real, or may be dismissed as spurious. Many filtering strategies are possible, but the simplest relies on counting the number of read pairs that support a deletion and only retaining those that exceed a minimum threshold of min_read_support supplied by the user. To reduce the number of false positive results we would like to remove from the list of reads supporting a deletion call those that are likely spurious. Since, low quality reads are already filtered by the Insert Size Filtering Service one of the few remaining sources of false positive signal are reads for which the inferred insert size is much larger than the actual deletion call that they span. These reads are either part of another, much larger, deletion, in which case they are accounted for in that deletion call, or they are spuriously mapped, as there is no other plausible reason for their large insert size. We thus, use a thresholding cutoff $\text{insert_size_threshold}$, supplied by the user, such that read pairs with $\text{insert_size} > \text{deletion_width} + \text{insert_size_threshold}$ are removed from the list of reads supporting a particular call. Afterwards, deletions that have read support exceeding the min_read_support threshold are selected as called variants.

The variant quality for the deletion is taken to be $v\text{qual} = \sum_{r \in R} r.\text{mapq}$ i.e. the Phred-scaled probability that all of the reads that support the deletion have been mis-mapped. All of the variants that have been called are emitted in the Insert Size Clustering Service's output stream in the format $M_{\text{del}} = \{m_i : m_i = (\text{header}, \text{payload})\}$ where $m.\text{payload} = V = \{v_i : v_i =$

$(s_id, contig, pos, length, vqual, R)\}$, as described in Table 5.12, where the header includes the query details that are being answered.

5.4 Rheos implementation

In order to prove the viability of the concepts behind Rheos we have built and tested a limited implementation of Rheos. This implementation provides a data streaming architecture of services and is able to perform genome alignment, followed by online germline SNP and deletion calling, as described in Section 5.3. The source code is available on github at - <https://github.com/llevar/rheos>. In this section we describe the details of the technical implementation, including the key approaches taken for the establishment of data streaming, service organization, scalable deployment, and performance optimization. Even though the implementation is fully functional and has been used to analyze real data, it is worth noting that due to the extremely broad scope of the overall Rheos framework, and the limited resources available, the implementation focuses on a narrow set of use cases and is meant to be treated as a proof of concept rather than a production system.

As in the implementation of Butler (see Chapters 3, 4) we make a concentrated effort to rely on established Open Source software frameworks where possible to ensure that components of Rheos are robust and scalable, require minimal maintenance, and can be easily deployed on a variety of platforms. Some of the key frameworks used by Rheos are Apache Kafka, Google Kubernetes, Docker, and Prometheus.

The implementation is focused on the following key services:

Read Mapping Service - Reads a FASTQ file and turns it into a set of streams for mapped and unmapped reads and read-pairs.

Locus Processor Service - Reads a stream of mapped read pairs and incorporates variant evidence from the reads into a model of genomic Loci. Emits a stream of updated Loci.

Locus Saver Service - Reads a stream of Loci and stores them in a distributed data store.

Variant Calling Service - Reads Loci from a data store and performs germline SNP calling. Emits a VCF file.

Insert Size Filtering Service - Reads a stream of mapped read pairs and emits a stream of filtered high quality read pairs that are discordantly mapped and can be used for calling deletions.

Insert Size Clustering Service - Reads a stream of discordantly mapped read pairs and calls germline deletions via KDE clustering. Emits a stream of deletion variants.

We first focus on the general architecture of a Rheos service and describe how services communicate with each other, then describe the deployment and operation of the Rheos system as a whole, and finally describe the individual services to comprise the Rheos implementation.

5.4.1 Rheos Service

A service in Rheos is a continuously running program with a well defined interface (implementing one or more operations described in Section 5.2.1), and a set of well understood operating characteristics. Even though a service author is completely free to use different technologies for different services (as long as the interface is respected), we have chosen to implement the initial set of Rheos services in Python. Since Rheos services are typically streaming services, their operations are most often invoked automatically when new data appears on the stream.

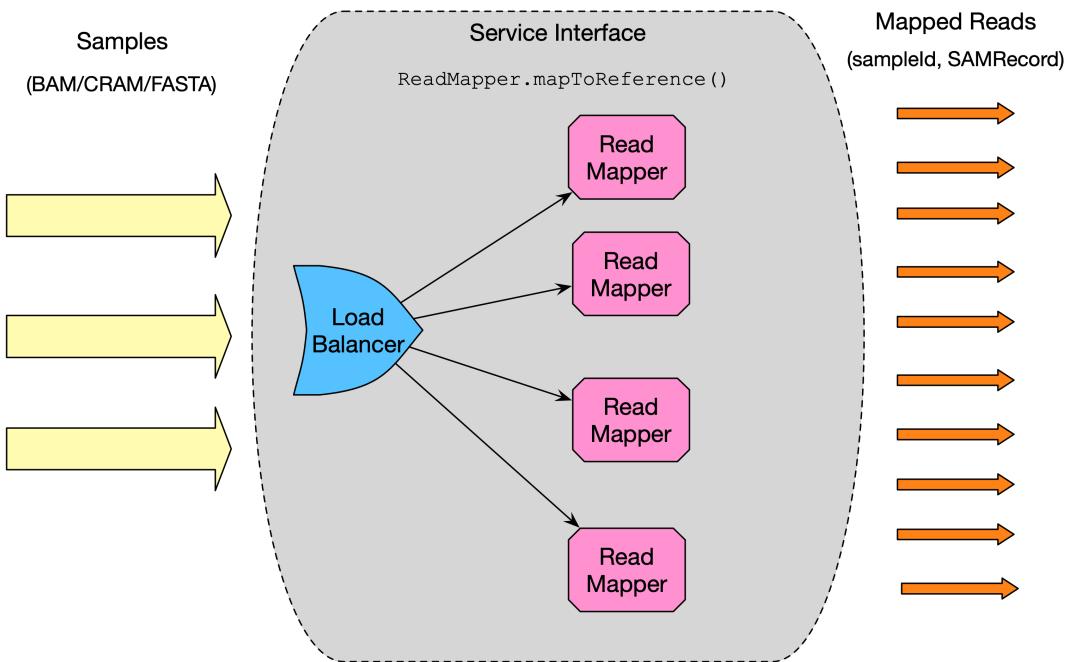


Figure 5.25: Several instances of the Read Mapping Service are managed behind a Load Balancer.

Figure 5.25 demonstrates several key concepts behind Rheos services using the example of a Read Mapping Service:

- The service transforms the granularity of the data from sample-level on the input to read-level on the output.
- The service implements a streaming operation of type Decorator, it decorates the incoming read data with additional information, that of the read coordinates, CIGAR string, mapping quality, etc.
- The service provides a uniform interface to its clients, that of *ReadService.mapToReference()*. Machines that implement this interface are indistinguishable from each other.
- Clients of the service talk to the service interface through the Load Balancer. The DNS entry for the service returns the IP of the Load Balancer which is then responsible for routing requests to one of the service instances. This allows one to control the scalability of the service, seamlessly adding and removing instances of the Read Mapper depending on the rate of the data

that is coming into the system, and the ability of downstream services to process it.

When services are deployed it is desirable to be able to easily deploy them to a variety of different environments and achieve a level of resource and environment isolation, so that other programs running on the same machine can only minimally impact the installation and running of the service. To accomplish these goals, all of the services in Rheos follow a containerized approach using Docker[152].

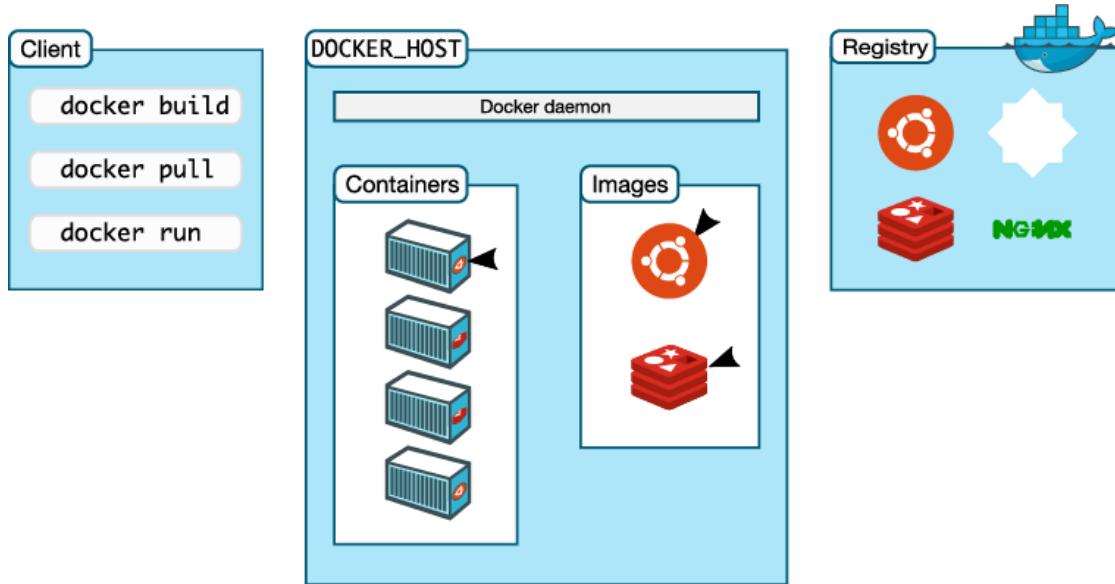


Figure 5.26: Docker system architecture (<https://docs.docker.com/engine/docker-overview/>).

Docker is an Open Source framework that isolates the execution of a program from the host machine and other programs by encapsulating it in a lightweight Linux container. The container is a separate instance of an operating system and a program running inside it is not aware of its enclosing environment. The Docker architecture consists of three main components (see Figure 5.26). A machine that can run Docker containers (the Docker Host) needs to be running the Docker Daemon. A container is a runtime instance of a Docker environment, but the blueprint from which all containers are created is called a Docker Image. The user starts with a base image that only has an operating system in it. The user can then augment the image by installing additional software packages and custom programs. This can be done interactively inside a shell inside a running container, or declaratively via a Dockerfile. When the user is finished preparing their image they can use the Docker Client to build it and upload it to an external Docker Registry. This is an external registry where various Docker images are hosted. When Docker Daemon needs to create a container it pulls down the requisite image from a registry to the local machine, and then instantiates the container. Containers can be started and stopped, created and deleted easily, and provide a great deal of abstraction and simplification when it comes to running many different applications, with potentially conflicting dependencies on a server. They also provide a seamless migration path from the local development machine, through the testing environment, and into production. All Rheos services are implemented to be encapsulated inside Docker containers.

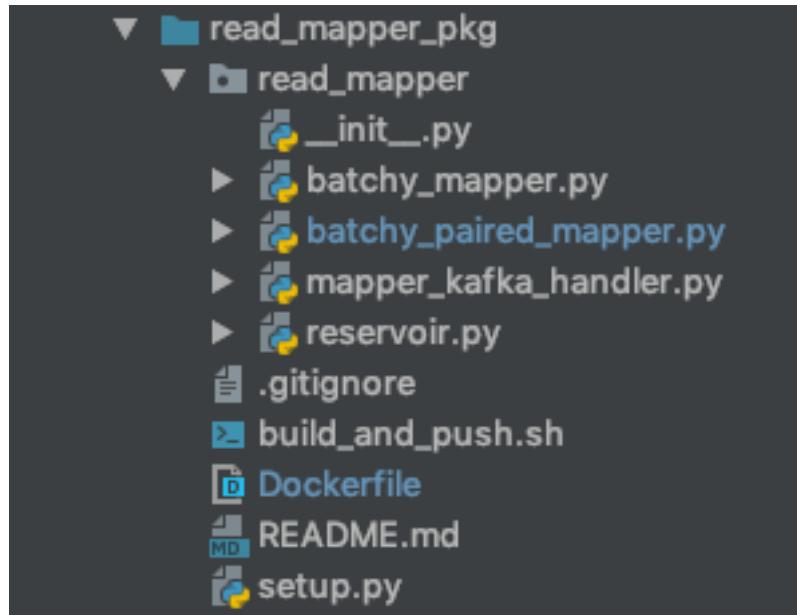


Figure 5.27: Rheos service file structure.

The layout of a Rheos service file structure is standard (see Fig. 5.27). One or more Python files implement the service functionality in the form of a Python module. The file *setup.py* provides a list of dependencies that are needed for the functionality of the service. The *Dockerfile* specifies how to build a Docker image of the service (See Listing 13).

Listing 13: Example Dockerfile for setting up a service.

```

1 FROM python:3.7.2-alpine3.8
2 RUN apk add --no-cache build-base python3-dev zlib-dev cython make bzip2-dev
   ↳ xz-dev libcurl
3 WORKDIR /app
4 COPY . /app/
5 RUN python setup.py install
6 RUN mkdir /app/log && touch /app/log/mapper.log
7 ENTRYPOINT ["python", "read_mapper/batchy_mapper.py"]

```

The Dockerfile uses a base image of Alpine Linux with python 3.7.2 preinstalled. Several addon linux packages are installed. The service module code is copied to the /app directory. *setup.py* is run to install all of the service's dependencies, and the program entrypoint for the Docker container is set to point to the service executable. The file *build_and_push.sh* is a shell script that is responsible for building a Docker image using the supplied Dockerfile and pushing this image to a Docker registry (hub.docker.com). Any deployment of the service can now pull down this image from the registry and instantiate new containers from it.

As there is a good deal of functionality that is common between several Rheos services there is a separate Python module called **rheos-common** that houses both utility classes and commonly used models (such as that for genomic loci) in its submodules. This module is hosted on Pypi and can be installed from anywhere using standard Python installation techniques such as *pip*.

The wire format for messages exchanged between services is currently schema-less and relies on standard Python serialization mechanisms (namely *pickle*, and *json*. Although *json* is completely generic, *pickle* is proprietary to Python and thus represents a weakness of the current implementation. A generic wire format with a defined schema will be a future improvement.

Serialized messages are streamed between services via distributed queues.

5.4.2 Queueing

For message queueing Rheos relies on an Open Source framework called Apache Kafka[109]. Kafka queues provide a distributed fault-tolerant implementation of the publish/subscribe mechanism, and streaming.

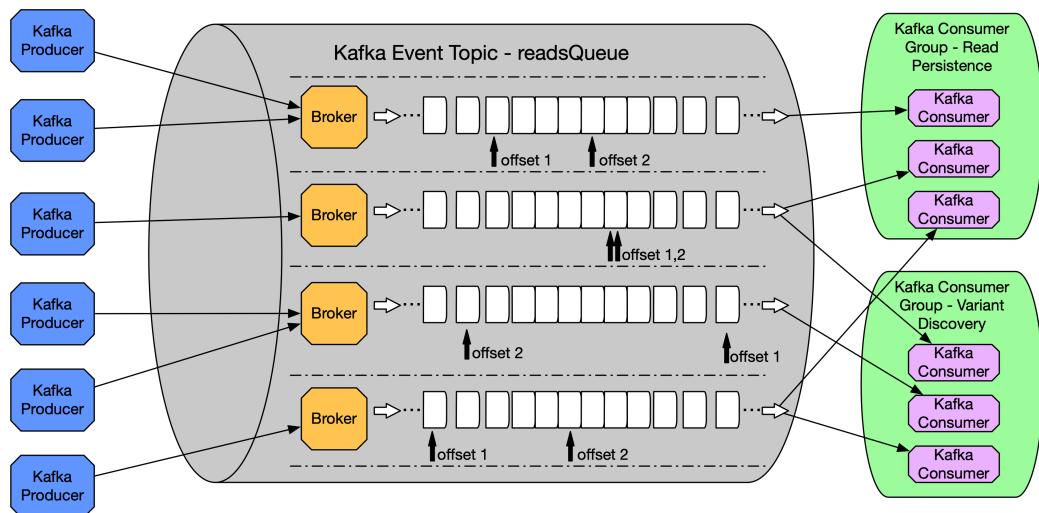


Figure 5.28: Example of a Kafka queue with multiple data producers and consumers.

At the heart of Kafka lies an Event Topic (see Figure 5.28). The topic is a distributed transaction log. When messages are put in the queue, they are never deleted by a consumer. Instead, consumers keep an offset into the log that tells them where they currently are in the processing of the data. Offsets can be moved freely so that a consumer may process the same message multiple times if needed. Data is deleted from the queue by virtue of a retention policy that removes data that is older than some specified time period.

The queue is distributed between several machines by using partitions. A Kafka partition is a function that specifies how many logical and physical partitions of the data exist. Data may be partitioned onto several machines based on data size considerations, or it may be partitioned based on a logical grouping. In the former case, an automated partitioning function is employed that decides which partition to send new messages to in round-robin fashion. In the latter case a logical partitioning function needs to be supplied by the user. Rheos mainly uses the latter approach to partition data by genomic coordinate (see Section 5.4.5 for details.)

Data producers put messages into the queue by communicating with a Broker. A Broker is a Kafka service that manages one or more queues on a particular machine. Multiple Brokers may be put in a cluster for load balancing. A data producer establishes a connection with a broker using a client library (`kafka-python` in the

case of Rheos). The partitioning function determines which machine a particular message ends up on.

On the data consumer side, consumers are organized into Consumer Groups. Each Consumer Group is a set of consumers that are assumed to be a separate entity from the consumers in all the other consumer groups. A Consumer Group has its own separate set of offsets into every partition of a topic that it is subscribed to and these do not interfere with the offsets of other groups. Assignment of consumers within a group to partitions of a topic can be automated or manual. In the case of automated assignment Kafka will decide on its own how to pair a consumer with a partition. In the case of manual assignment the partitioning function is used to make the assignment. It is in general not advisable to have multiple consumers from the same group reading data from the same partition.

We implement a generic Kafka Handler in the `rheods-common` module. This handler allows Rheos services to initialize Kafka producers and consumers, serialize and deserialize data using pickle and json, and to store and retrieve messages using the topics that have been set up for inter-service communication.

5.4.3 Partitioning

Partitioning is important in Rheos not only because the data is quite big and thus needs to be parceled out to multiple machines, but also because data exhibits strong locality subsequent to the read mapping stage, where most of the data that is required to make decisions about the presence or absence of a genomic variant resides in some small genomic neighbourhood around the variant. It is thus natural for us to partition the read data (which is biggest in size) by genomic coordinate.

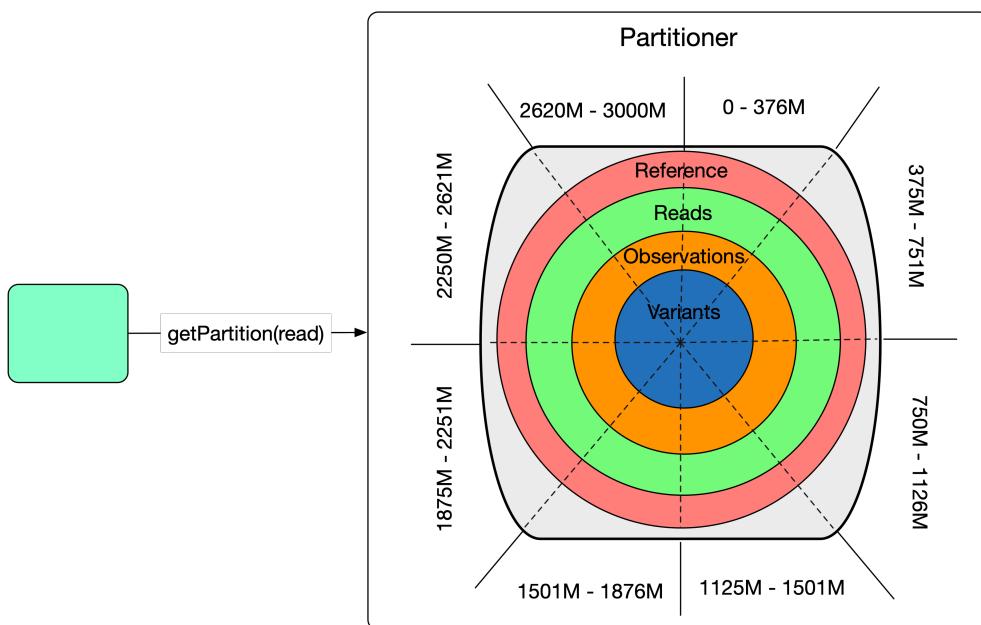


Figure 5.29: Rheos data is partitioned by genomic coordinate.

Partitioning can be challenging because different partitions end up on different machines, yet certain reads and structural variants end up spanning one or several partitions. We need to decide what to do at the partition boundaries, i.e. if a

particular read ends up spanning the boundary, does it end up in one partition, the other, or both? If a read ends up in two partitions how does one avoid double counting the read in downstream processing? Likewise, if the reads supporting one breakpoint of a structural variant lie in one partition, and the reads supporting the other breakpoint lie in another partition, how can these be put together to form a single variant?

We take a simple approach in the initial Rheos implementation that will require further work when the system is elaborated. The following assumptions are made by the Partitioner:

- All partitions are of equal size, except at the end of contigs.
- Reads that span multiple partitions are assigned to all partitions that they map to (see Figure 5.30).
- A given read will always map to the same set of partitions.
- All reads that are unmapped end up in the "unmapped" partition.
- All reads that map to special contigs (like decoy sequences) are mapped to a special "other" partition.

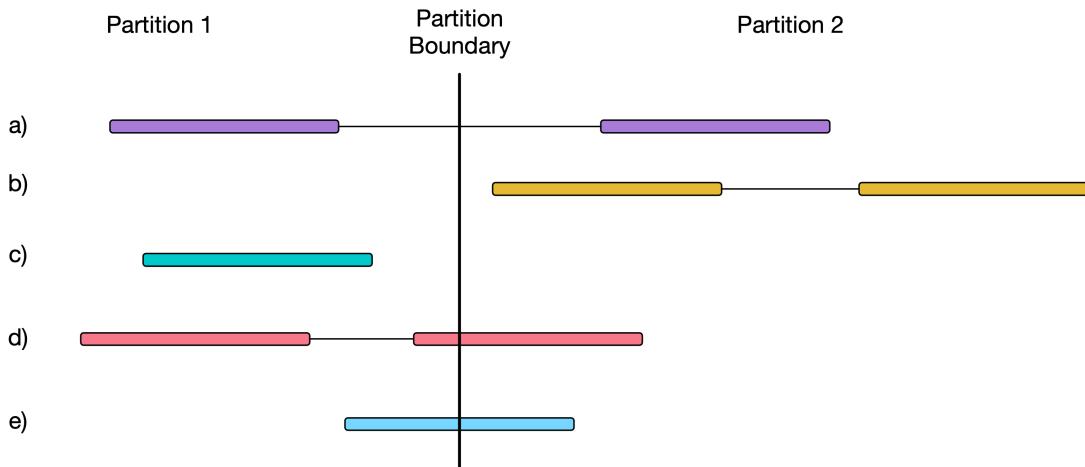


Figure 5.30: Reads that map near the partition boundary - a) The read pair maps to partitions 1 and 2, b) The read pair maps to partition 2 only, c) The single read maps to Partition 1 only, d) The read pair maps to partitions 1 and 2, e) The single read maps to partitions 1 and 2.

In general the partitioning problem can be stated as follows:

Given a reference genome $G = G_{main} \cup G_{other}$, where $G_{main} = \{g_i : g_i = (name_i, length), name_i \in \{1, \dots, 22, X, Y, MT\}\}$ and $G_{other} = \{g_i : g_i = (name_i, length), name_i \in Aux_Contig_Names\}$, we assign a set of indexed partitions $P = \{(g_i.name, start, end, i) : g_i \in G_{main}, i \in [1, |G_{main}|]\} \cup \{("other", |G_{main}| + 1), ("unmapped", |G_{main}| + 2)\}$, where given a set of reads $R = \{r_i : r_i = (contig, pos, end)\}$ (most often a read or a read pair), we desire to have a function $f : R \mapsto \mathcal{P}(P)$ (where $\mathcal{P}(P)$ is the power set of P) that maps those reads to a set of partitions from P .

We opt for partitions of fixed width partition_width (except at ends of contigs) and implement Algorithm 8, that, given G , produces P , and for a given interval or read pair returns a set of partitions they map to:

Algorithm 8: Partitioning the reference genome by a fixed width and mapping reads to partition sets.

```

Function INITIALIZEPARTITIONER( $g_{\text{main}}, g_{\text{other}}, \text{partition\_width}$ ) begin
     $\text{partition\_index} \leftarrow 0$ 
    for  $\text{name}, \text{length}$  in  $g_{\text{main}}$  do
         $\text{num\_partitions} \leftarrow \text{CEILING}(\text{length}/\text{partition\_width})$ 
        for  $i \leftarrow 0$  to  $\text{num\_partitions}$  do
             $\text{left} \leftarrow i \times \text{partition\_width} + 1$ 
             $\text{right} \leftarrow \text{MIN}((i + 1) \times \text{partition\_width}, \text{length})$ 
             $\text{new\_record} \leftarrow \text{PARTITIONRECORD}(\text{name}, \text{left}, \text{right}, \text{partition\_index})$ 
            // Create a member of  $P$ 

            ADDBREAKSBYCONTIG( $\text{name}, \text{left}$ ) // Keep a hashmap of partition boundaries by contig

            ADDPARTITIONSBYCONTIGANDLEFT( $\text{partition\_index}, \text{name}, \text{left}$ ) // Keep a hashmap of partition records by contig and left boundary.

             $\text{partition\_index} \leftarrow \text{partition\_index} + 1$ 
    
```

```

Function GETPARTITIONSFORINTERVAL( $\text{contig}, \text{left}, \text{right}$ ) begin
     $\text{partitions\_for\_int} \leftarrow \text{SET}()$  // Partition records are a set, to avoid duplicates

    if  $\text{contig} \notin \text{GETOTHERCONTIGNAMES}()$  then
         $\text{breaks} \leftarrow \text{GETSORTEDBREAKSBYCONTIG}()$  // Breaks are sorted by left coordinate

         $\text{left\_index} \leftarrow \text{BINARYSEARCH}(\text{breaks}, \text{left})$ 
         $\text{left\_partition} \leftarrow \text{GETPARTITIONBYCONTIGANDLEFT}(\text{contig}, \text{breaks}[\text{left\_index} - 1])$ 
        ADDTOSET( $\text{partitions\_for\_int}, \text{left\_partition}$ )
         $\text{right\_index} \leftarrow \text{BINARYSEARCH}(\text{breaks}, \text{right})$ 
         $\text{right\_partition} \leftarrow \text{GETPARTITIONBYCONTIGANDLEFT}(\text{contig}, \text{breaks}[\text{right\_index} - 1])$ 
        ADDTOSET( $\text{partitions\_for\_int}, \text{right\_partition}$ )
    else
        ADDTOSET( $\text{partitions\_for\_int}, \text{GETOTHERPARTITION}$ )
    
```

```

Function GETPARTITIONSFORREADPAIR( $\text{read\_pair}$ ) begin
     $\text{partitions\_for\_reads} \leftarrow \text{SET}()$ 
    for  $\text{read}$  in  $\text{read\_pair}$  do
        if  $\text{read.mapped}$  is TRUE then
             $\text{left} \leftarrow \text{read.pos} + 1$ 
             $\text{right} \leftarrow \text{read.end}$ 
             $\text{contig} \leftarrow \text{read.contig}$ 
            ADDALLTOSET( $\text{partitions\_for\_reads}, \text{GETPARTITIONSFORINTERVAL}(\text{contig}, \text{left}, \text{right})$ )
        else
            ADDTOSET( $\text{partitions\_for\_reads}, \text{GETUNMAPPEDPARTITION}$ )
    
```

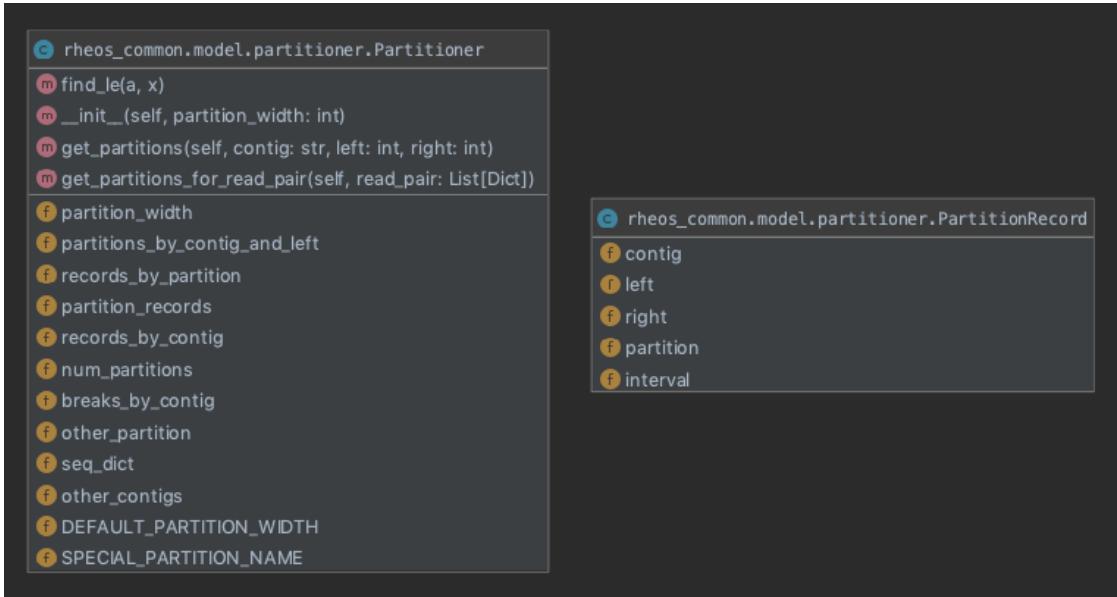


Figure 5.31: Rheos Partitioner class diagram.

Algorithm 8 is implemented in Rheos via the `Partitioner` and `PartitionRecord` classes of the `rheos-common` package. Namely, members of P are implemented as instances of `PartitionRecord`, function `INITIALIZEPARTITIONER(g_main , g_other , $partition_width$)` is implemented in `Partitioner.__init__()`, function `GETPARTITIONSFORINTERVAL($contig$, $left$, $right$)` is implemented in `Partitioner.get_partitions()` method, and function `GETPARTITIONSFORREADPAIR($read_pair$)` is implemented in `Partitioner.get_partitions_for_read_pair()`.

In our tests we have been using a partition width of 2×10^7 bases, resulting in 148 partitions for read data. The Partitioner is used in the Read Mapping Service to determine which partition to write data to, as well as in the Locus Processor Service to determine which partition is assigned to which instance of the service.

5.4.4 Deployment

Deployment of large-scale distributed systems can be complex as each component can have its own requirements related to computational resources, environment, or the ability to scale. This heterogeneity of requirements demands a flexible deployment approach that can cater to the individual needs of the various components of the system. Additionally, there is a wide variety of computational infrastructures into which a user may wish to deploy the system, again with a distinct set of capabilities and limitations. With Butler (see Chapters 3, 4) we solve this problem by utilizing a separate framework (Terraform) for abstracting the specifics of cloud provider's provisioning API, another separate framework (Saltstack) for providing a flexible configuration management facility that can configure software on a variety of platforms, and a separate suite of frameworks (InfluxDB, Grafana, Telegraf, Kapacitor) for operational management of the deployed components. We take a somewhat different approach with Rheos, one that based on the containerized nature of Rheos services, and one that relies on a single common execution environment being de-

ployed on top of any cloud provider, or other types of computational infrastructure. The environment that we use is Google Kubernetes[22].

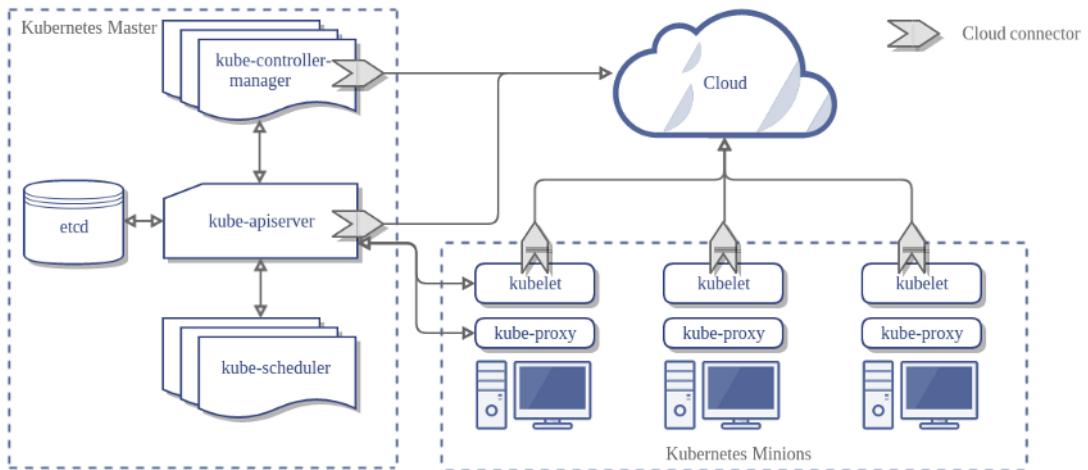


Figure 5.32: Kubernetes high level architecture (<https://kubernetes.io/docs/concepts/architecture>).

Kubernetes is a cluster resource management and container execution framework developed by Google. It can deploy on virtually any cloud as well as bare metal hardware (which makes it more flexible than the approach adopted in Butler). A Kubernetes cluster is a collection generic compute hosts (Nodes) each running the `Docker` and `kubelet` daemons. These nodes go into a generic pool of computational resources, although nodes with specialized capabilities, such as high RAM, or advanced GPUs can be held in separate pools. A master node is responsible for keeping track of the configuration and computational capabilities of the cluster by virtue of an etcd registry that it keeps. This node runs a special management daemon called `kubeadm` and schedules the deployment and execution of workloads on the cluster, all of which are encapsulated in Docker containers. See Figure 5.32 for the general high level architecture of Kubernetes. This architecture is scalable to over 2500 compute nodes per cluster[8].

In order to facilitate inter-container communication Kubernetes provides an overlay network that is responsible for managing address translation, network security, and local DNS by running a proxy agent `kube-proxy` on each node. This allows various components of the system to refer to each other by name rather than IP address, and enables role-based access to various network resources.

Kubernetes was built for deploying and operating services and provides a large number of high-level abstractions to facilitate the full service lifecycle. Kubernetes objects are managed through declarative documents in YAML format that describe the desired state of the system. The user interacts with the cluster by virtue of a CLI (`kubectl`) running on the master node, or via a GUI dashboard (see Figure 5.33).

At the lowest level of abstraction in Kubernetes hierarchy is a Pod. The Pod is a wrapper around a Docker container and describes some useful metadata for categorizing as well as describing the resource requirements of the underlying container that are used for scheduling the container onto nodes. These resources include CPU and memory requirements, as well as requests for various types of storage (called



Figure 5.33: Kubernetes dashboard.

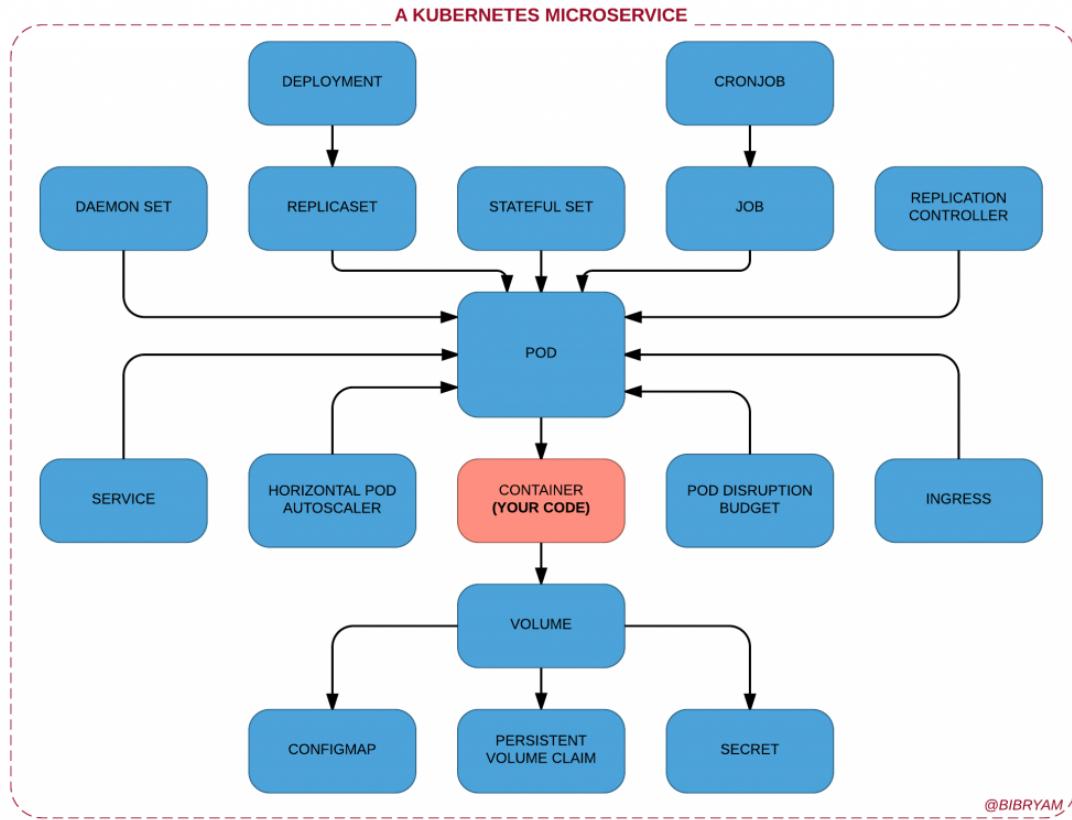


Figure 5.34: Kubernetes microservice components (from [9], by Bilgin Ibryam)

a PersistentVolumeClaim). A Pod is scheduled for execution only when the cluster manager can find a suitable node that can meet all of the pod's resource requirements. If the Pod exceeds its stated resource requirements during its lifetime it will

be terminated by the scheduler.

Since the goal of kubernetes is to enable computation at scale, Pods are rarely run as single instances. Depending on the intended usage of the underlying container Pods can be organized in groups such as a DaemonSet, ReplicaSet, StatefulSet, Job, etc. These abstractions describe the runtime behaviour of groups of Pods and provide lifecycle management support for each. For instance, a ReplicaSet provides a group stateless interchangeable Pods where each Pod can identically service a request from a client, and Pods can be scaled horizontally. The configuration for a ReplicaSet may specify the number of such Pods that should be running at a given time and the scheduler will create and delete Pods as necessary to make sure that the requisite number of Pods is running. Other groups of Pods that do not scale horizontally, and require access to non-ephemeral secondary storage (such as those running a database) may use the StatefulSet to provide such services. A DaemonSet makes sure that every node in a cluster is running an instance of a particular Pod. A Service is a higher level abstraction that provides a logical grouping of pods and provides a number of capabilities such as DNS, load balancing, and security.

In Rheos we make use of Kubernetes for two purposes - one is to deploy and configure the various products that Rheos services rely on for their operation. These include:

Kafka - The queueing framework that all Rheos services use for communicating is deployed on Kubernetes as a set of fault tolerant Docker containers.

Redis - Is a scalable in-memory key-value data store that Rheos services use to store intermediate results.

PostgreSQL - Is an SQL database used for persisting variants to disk.

Prometheus - Is a monitoring and alerting framework used to maintain operational control of the running system.

Grafana - Is a metrics dashboarding software used for visualization of health metrics.

Elasticsearch - Is an indexing product used for harvesting and storing application logs.

Jenkins - Is an automated build and deployment tool used for creating software builds.

Kubernetes Dashboard - Is a UI used for managing Kubernetes clusters.

For each product we maintain a set of configuration files used for Kubernetes deployments and configuration of these tools (see Figure 5.35).

Additionally we use Kubernetes to configure and deploy the services of Rheos themselves, organized as fleets of Docker containers. This allows us to enjoy the flexibility of being able to deploy on virtually any hardware, the environment isolation provided by Docker, and the resource management and elastic scalability afforded by Kubernetes, which are all key concerns when operating a large-scale service-oriented system.

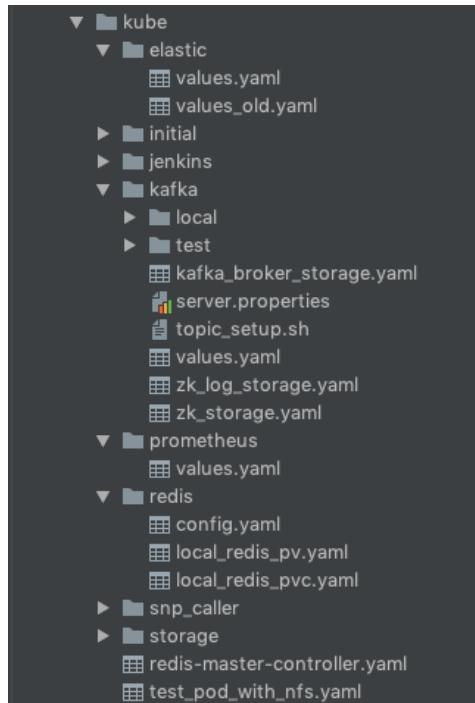


Figure 5.35: Rheos Kubernetes configurations file layout.

5.4.5 Services of Rheos

We build on top of the concepts of a general Rheos Service, Kafka topics, the Kubernetes framework and other products described earlier in this section to provide a cohesive implementation of a set of services that implement two major end-to-end use cases encountered in genomic analysis - germline SNP calling, and germline SV (deletion) calling. As Rheos matures these services will be extended to provide enhanced functionality and increased scalability required for production applications of the framework.

Read Mapping Service

The Read Mapping Service is the entrypoint of the initial Rheos implementation. Although future versions of Rheos will likely have a different entrypoint, namely a Read Streaming Service that will accept data from an external endpoint, such as a data repository or directly from the sequencer, we proceed directly to read mapping in this initial implementation as a matter of simplification.

This service aims to implement two read mapping operations specified in Section 5.3.2 - *mapToReference()* and *mapPairToReference()*. It reads a user-specified FASTQ file (the sample), as well as a genome reference file from disk and emits four data streams *mappedReads*, *mappedReadPairs*, *unmappedReads*, and *unmappedReadPairs*. These operations allow the service to deal with both paired and unpaired short reads produced by Illumina sequencers. The partitioning strategy of Section implemented in the `rheos-common` package is used to determine which partition of their respective topics reads end up in. Instead of implementing our own read mapper from scratch we are making use of a python module called `mappy`, which is an interface to the popular read alignment tool `minimap2`[127], by Heng Li, which is based on the hashmap approach to alignment discussed in Section

2.2.2. The reason we chose to provide a wrapper rather than a full implementation is that minimap2 already has an API that supports a read-by-read alignment that can be adopted for a streaming use case. Additionally, minimap2 is known to be accurate, fast, and supports mapping of long reads, and other use cases, making it an attractive framework to work within for read alignment.

Although the Read Mapping Service can map and stream individual reads, this approach proves quite impractical. In our tests of performance, we observed that for a single read the cost of serializing the mapped read data to send it over the wire was nearly 40% of the overall processing time. To reduce the proportion of computational resources used for serialization/deserialization we implemented a read batching strategy based on the reservoir concept, that groups similar reads together and sends them over the wire as a group. A `PartitionedReservoirSet` that implements this functionality is shown in Figure 5.36.

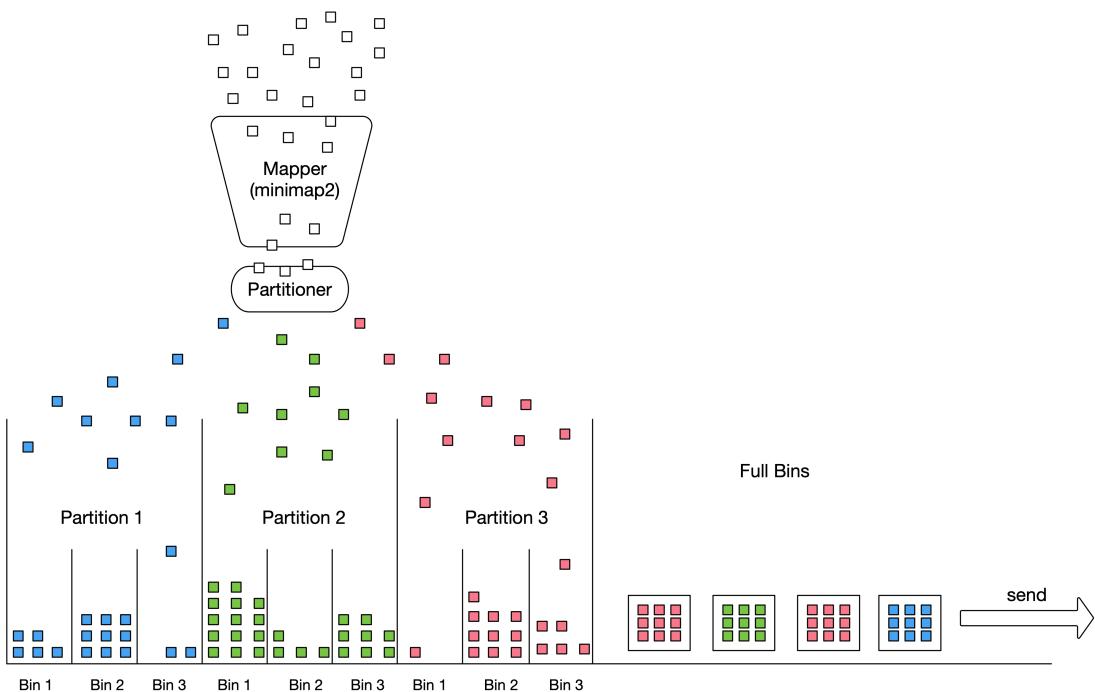


Figure 5.36: A `PartitionedReservoirSet` is used to group together similar reads before streaming them as a mini batch.

Reads are read one-by-one from a FASTQ file. Each read is attempted to be paired. If no pair exists for the read it is put into an accumulator (a type of hashmap). If a read completes a read pair, it is popped from the accumulator and the read pair is sent through to alignment by minimap2. Once the read pair is aligned it is put through the Partitioner to assign a set of PartitionRecords to it. Each PartitionRecord is further subdivided by genomic interval into a set of bins that act as reservoirs. The size of the bins is controlled by user supplied parameters where `num_bins` specifies how many bins are created per partition, and `bin_size` specifies capacity in read pairs. Each `Reservoir` accumulates read pairs that map within its genomic region, until reservoir capacity is reached. At that point all of the read pairs from that reservoir are serialized and emitted from the service as a single message. The reservoir is then emptied and is ready to accept new elements. When a whole sample is finished alignment all of the non-empty bins are serialized and emitted by

the service as individual messages. Figure 5.37 shows the classes and methods of the Read Mapping Service and the `PartitionedReservoirSet`.

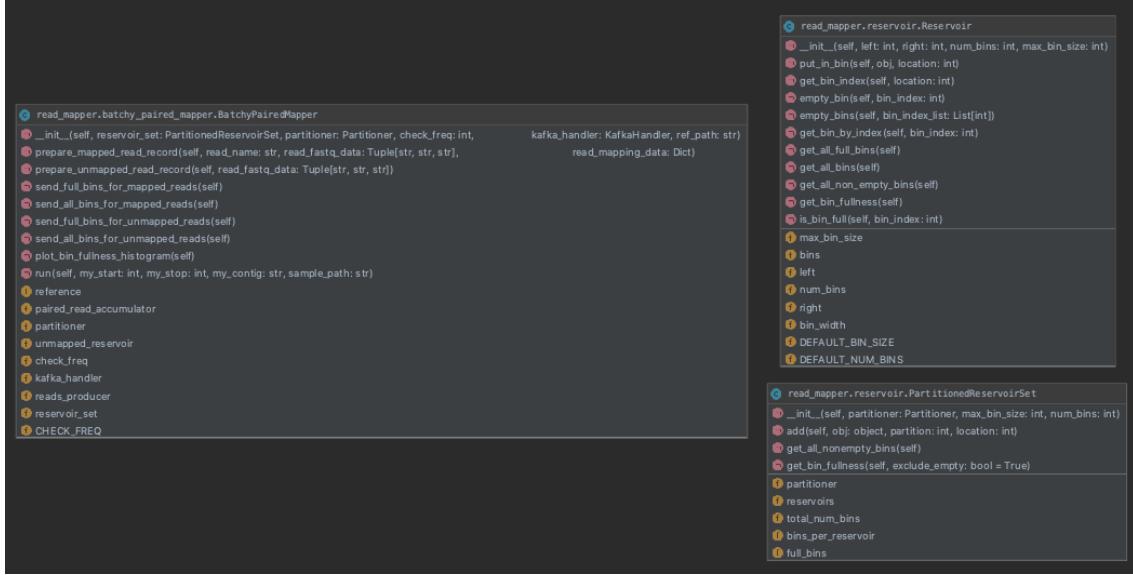


Figure 5.37: Classes of the `ReadMappingService`.

The disadvantages of the mini-batching approach are the decreased initial throughput (no data is emitted by the service before at least one of the reservoirs fills up), and increased memory footprint of the Read Mapping Service. The reservoir approach, however, reduces the proportion of the overall processing time due to serialization by over 90%. Additionally, there are benefits to processing the read data in small batches of closely-mapped reads by downstream services, because all of the loci that need to be updated by the reads can be fetched into memory and efficiently processed, and since the reads map closely together, fewer total loci are updated by such a batch than by a random collection of reads. In general, since the usage of the `PartitionedReservoirSet` can be controlled by the user via service parameters, there is a good degree of flexibility when it comes to trading off these performance considerations.

Locus Processor Service

The Locus Processor Service is a Rheos service that listens to the stream of mapped read pairs produced by the Read Streaming Service and updates a set of genomic loci that the reads overlap with the variant evidence contained in the reads. It then emits a stream of updated loci that can be further processed by downstream services for variant calling. Figure 5.38 shows a schematic view of the Locus Processor Service. The Locus Processor Service is of Local State Aggregator type and implements the iterative Bayesian inference framework described in Section 5.3.3, namely it implements the stream operation `updateLociFromRead()`, Equations 5.10, and Algorithms 6, and 7. The service uses the same Partitioner to decide where to get data from as is used by the Read Mapping Service.

A single message received by the Locus Processor Service represents a full bin of mapped read pairs from a single partition of the `mapped_read_pairs` kafka topic. `BatchyPairedLocusProcessor.process_messages()` is the main processing loop of the service (see Figure 5.39). An in-memory collection of `Locus`

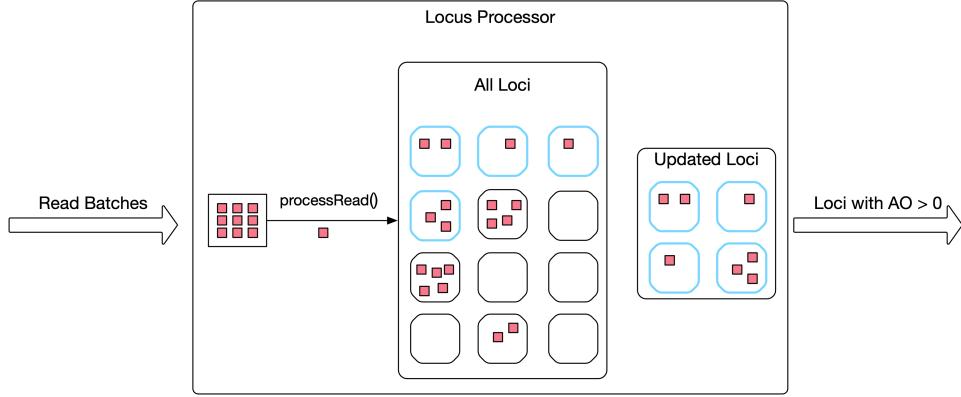


Figure 5.38: The Locus Processor Service uses read data to update a set of loci.

objects for the assigned partition, organized in a hashmap by genomic coordinate is the main holder of state for the service. For each received message, the `BatchyPairedLocusProcessor.process_reads()` method is responsible for processing all of the contained reads. It acts as the implementation of Algorithm 6. Namely, for each read, the algorithm simultaneously walks the genome reference and the read CIGAR string looking for matches. Different CIGAR elements consume the reference, the read, or both. For instance, a deletion consumes the reference sequence, but not the read. It's the opposite case for an insertion, and a sequence match consumes both. Deletion and insertion CIGAR elements will in the future be used for indel variant calling, but at the moment they are simply passed over. When a match CIGAR element is found, the `BatchyPairedLocusProcessor.handle_match()` method is called.

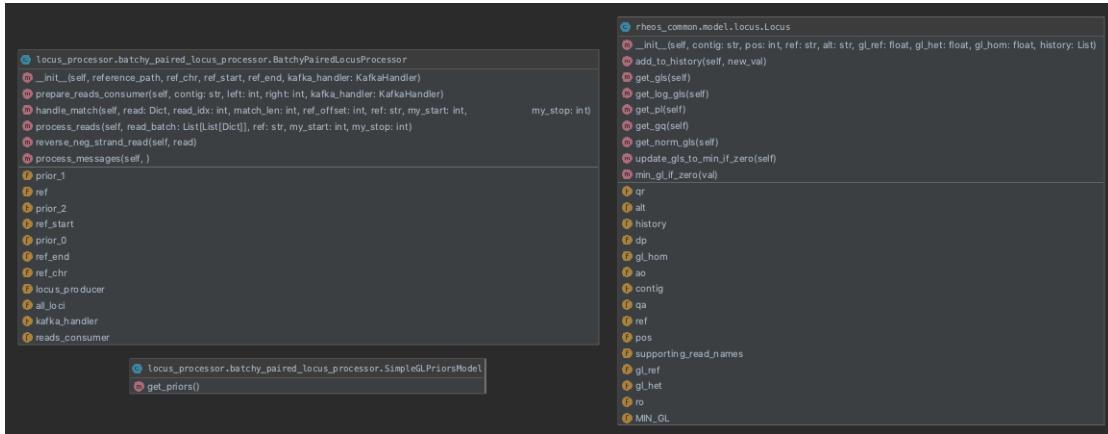


Figure 5.39: The Locus Processor Service classes and methods.

This method implements Algorithm 7. For each match the set of `Locus` objects overlapping the match are retrieved. The model of each `Locus` is updated using Equation 5.10, depending on whether that position in the read matches the reference. Reads that map to the negative strand are reverse complemented using `BatchyPairedLocusProcessor.reverse_neg_strand_read()` and matched from the end.

For each input message received, all `Locus` models that have been updated by reads in this message, and that potentially have variants (determined by `Locus.AO > 0`,

where AO is the count of alternative allele observations for that locus) are bundled together and emitted as a single message of updated loci from the Locus Processor Service.

Locus Saver Service

The Locus Saver Service is a fairly simple service that acts as an aggregating and persistence mechanism for Locus objects. Rheos uses an open source in-memory distributed key-value store called Redis[29]. Using this product allows Rheos to keep a large number of Locus objects in memory for fast retrieval, while offering options for distributed access and optional disk persistence.

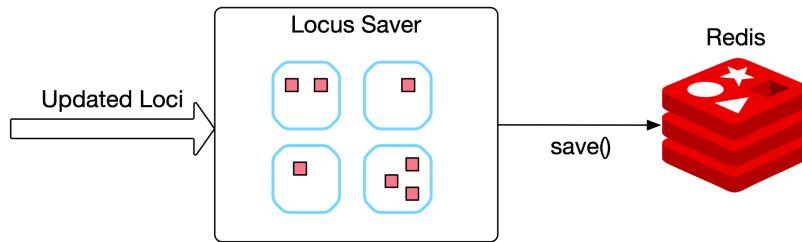


Figure 5.40: The Locus Saver Service data processing schematic.

The Locus Saver Service receives messages from the *snp_loci_batchy* kafka topic, where each message contains a batch of updated Locus objects. The service writes all of the updated loci to Redis potentially overwriting any existing values.

Variant Calling Service

The Variant Calling Service is not a stream-based service, instead it provides a querying interface into Rheos via its *callVariants(region, calling_threshold)* operation, where the user provides a region of interest and variant quality threshold and the service creates a VCF file of all called variants in the region that meet the variant quality criteria.

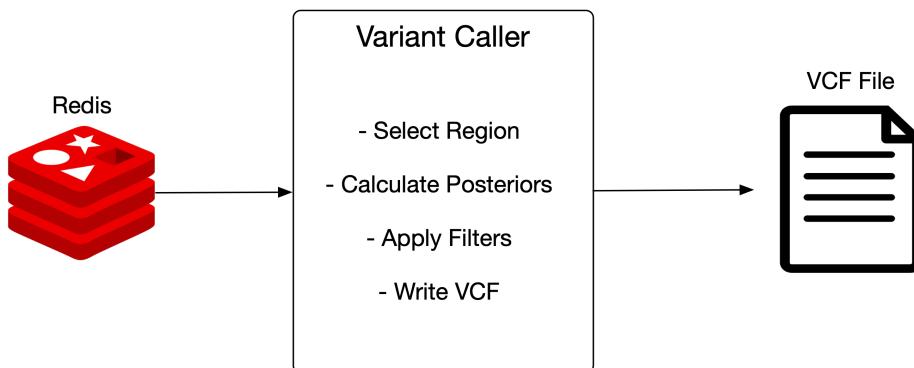


Figure 5.41: The Variant Calling Service data processing schematic.

This service communicates with the Redis locus store (see Figure 5.41) and retrieves all of the loci that match the query region. For each locus it calculates the posterior genotype probabilities via Equation 5.11. Variants that exceed the user provided

quality threshold will be written to the output VCF file. It is possible to apply other variant filtering criteria at this stage to improve calling stringency. As the service requires access to secondary storage it is deployed as a StatefulSet under Kubernetes.

Insert Size Filtering Service

The Insert Size Filtering Service is a streaming service of Filter type. It implements the *filterDiscordantPairs()* operation in order to obtain a stream of read pairs that are fit for germline deletion calling as described in Section 5.3.4. The service initially only accumulates read pairs in order to estimate the insert size distribution. The default initial sample size for this purpose is 100000 read pairs, but this value is controlled by the user. Using this sample we estimate the median insert size, and subsequently the Median Absolute Deviation of the insert size which acts as our cutoff for discordantly mapped reads. Once the estimate is obtained we send the accumulated reads through the rest of the filtering process, followed by any other new read pairs that are observed in the data stream. Read pairs that pass filtering are sent to the *discordant_read_pairs* kafka topic.

Insert Size Clustering Service

The Insert Size Clustering service implements the *addReadPair()* streaming operation and the *callDeletions()* querying operation. *addReadPair()* observes the *discordant_read_pairs* data stream and adds any messages that are received into the internal data structures used for deletion calling.

```
C insert_size_clusterer.insert_size_clusterer.InsertSizeClusterer
(m __init__(self, kafka_handler: KafkaHandler)
(m get_insert_size(self, read_pair: List)
(m make_interval_tree(self, records)
(m prepare_sorted_records(self, reads)
(m get_estimated_model_and_density(self, my_data)
(m get_cluster_locations(self, estimated_density)
(m get_deletion_boundaries(self, intervals)
(m get_supporting_reads(self, intervals, left, right, records, return_filtered=False)
(m generate_plots(self, records, estimated_density, cluster_locations)
(m print_hits(self, hits)
(m run(self)
(f reads_source
(f kafka_handler
(f deletion_producer
```

Figure 5.42: The Insert Size Clusterer class methods.

The reads are stored as both a List for general handling and clustering, as well as an Interval Tree (see Figure 5.22) - a type of balanced binary search tree useful for finding interval overlaps. We use an open source implementation of the Interval Tree from <https://github.com/chainleib/intervaltree>.

When *callDeletions()* is executed the accumulated reads are prepared for processing by finding their center and insert size. The read centers are then clustered using the *KernelDensity* object from the `sklearn.neighbors` Python package (see Figure

5.23). Peaks of the estimated distribution are found using the `argrelextrema()` function from the `scipy.signal` package. Given the location of a cluster center we find the set of read pairs that overlap the center via the Interval Tree. Given a pre-constructor Interval Tree t and a cluster center coordinate $center$ we obtain the necessary set of intervals by simply calling $overlaps = t[rec.center]$.

For each set of overlap intervals the deletion size is set to be the maximum common overlap between the inner sizes of all such intervals (see Figure 5.24). All overlapping read pairs whose $insert_size > deletion_size + insert_size_threshold$ are removed from the list of reads that support the deletion, and if the number of remaining reads is $> min_support_threshold$ the deletion is called as real.

5.5 Experimental Validation

We focused on two primary use cases in validating the initial implementation of the Rheos framework. The first is the ability of the framework to accurately perform germline SNP calling. The second is the ability of the framework to accurately perform germline deletion calling. Deployment of the framework onto cloud computing infrastructure was also tested. Detailed performance testing was not performed because of the limited scope of the project.

5.5.1 Deployment

The Rheos framework has been deployed onto the academic cloud computing infrastructure provided by EMBL/EBI Embassy Cloud, an OpenStack environment with 164 CPUs, 744 GB RAM, and 15 TB of block volume storage. Resources were split into 12 virtual machines with the following variety of characteristics:

Table 5.13: Rheos deployment infrastructure

Count	CPU	RAM	Ephemeral Disk
1	4	8GB	60GB
4	8	32GB	100GB
6	16	64GB	100GB
1	32	224GB	100GB

One VM with 4 CPU and 8 GB RAM was designated as the Kubernetes master node, with other small services like the Kubernetes Dashboard and Weave Networking Dashboard deployed to it as well. Other VMs were put into the pool of worker nodes with automated deployment scheduling of Docker containers provided by Kubernetes. A Weave overlay network was set up between all of the nodes in the Kubernetes cluster, creating a private subnet for inter-VM and inter-container communication but inaccessible from the outside.

The following products were set up to support the running of Rheos:

Kafka - A Kubernetes Kafka service was deployed with a redundancy of 3, deploying 3 brokers on 3 separate nodes, to ensure availability and scalability.

Zookeeper - A Kubernetes Zookeeper service was deployed with a redundancy of 3.

Redis - A Kubernetes Redis server was set up with 1 master and 1 replication slave.

Prometheus - Prometheus was set up for monitoring with 1 master, and an agent running on every node.

Grafana - Grafana was set up as a single container for monitoring dashboarding.

Elasticsearch - Elasticsearch was set up as a single server for collection of application logs.

Jenkins - Jenkins was set up as a single server for automated deployment and continuous integration of new builds in the testing environment.

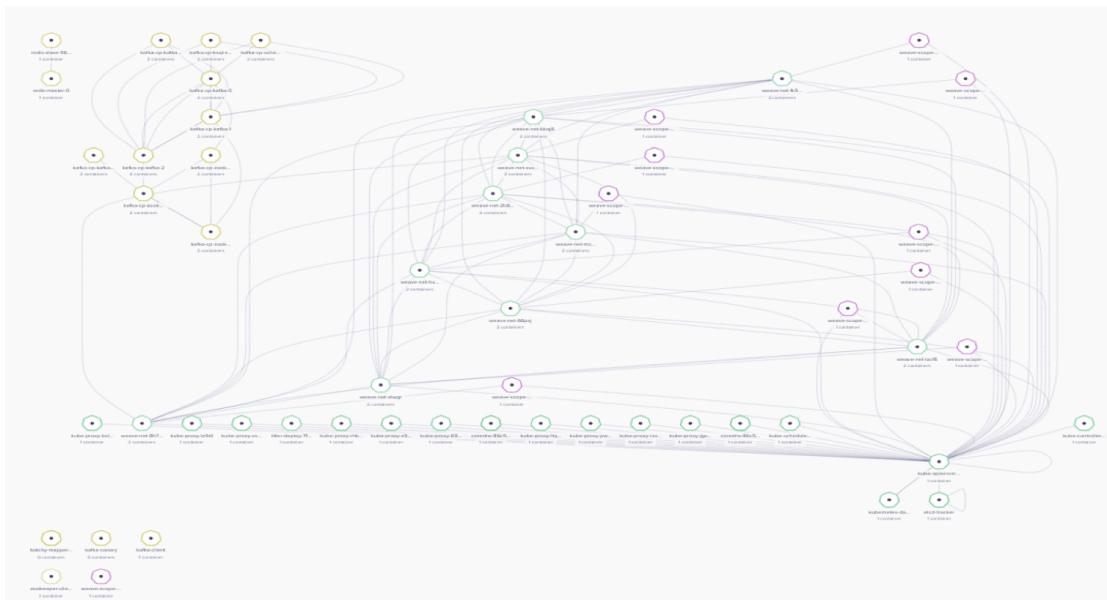


Figure 5.43: Docker containers that are part of the test Rheos deployment.

The services of Rheos were deployed into the Kubernetes environment in the following configuration:

Read Mapping Service - The service was deployed as a set of 2 Kubernetes Pods on separate nodes with access to a 500TB NFS share for reading sample data, and a local copy of the human reference genome, preprocessed for use by minimap2.

Locus Processor Service - The service was deployed as a set of 6 Kubernetes Pods, each with a local copy of the human reference genomes in FASTA format.

Locus Saver Service - The service was deployed as a set of 2 Kubernetes Pods on separate nodes.

Variant Caller Service - The service was deployed as a single Kubernetes Pod, with access to a 500TB NFS share for writing the VCF file.

Insert Size Filtering Service - The service was deployed as a single Kubernetes Pod.

Insert Size Clustering Service - The service was deployed as a single Kubernetes Pod.

The following Kafka topics were set up for management of in-flight data. A replication-factor of 1 was used.

Table 5.14: Rheos Kafka topics.

Name	Partitions
<i>mapped_readpairs_batchy</i>	168
<i>unmapped_readpairs_batchy</i>	2
<i>mapped_readpairs_secondary_batchy</i>	10
<i>snp_loci_batchy</i>	2
<i>discordant_read_pairs</i>	1
<i>deletions</i>	1

Figure 5.43 shows the set of Docker containers that were used in the Rheos Kubernetes deployment. In addition to the Kubernetes test environment, local testing of individual components was performed on a MacBook Pro 2018, with a 6-core 2.9 GHz Intel Core i9 processor, and 32 GB RAM.

5.5.2 Sample Selection and Preparation

To assess the ability of Rheos to perform germline SNP and SV calling we have selected a well characterized genomic sample from the Genome in a Bottle Consortium[245], namely the NA12878 sample that was first sequenced in the context of the 1000 Genomes Project, but has subsequently been sequenced by a variety of orthogonal sequencing technologies to very high levels of coverage and is very well studied. Choosing this sample allows us to work with data that is known to be of high quality and one that has a high quality set of published genomics variants to compare Rheos calls against.

We downloaded the sample RMNISTHS_30xdownsample.bam a 147GB file from the GIAB FTP repository at ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/NA12878/NIST_NA12878_HG001_HiSeq_300x/. This sample is a high coverage 300x sequencing run of the Illumina HiSeq 2500 instrument using a 2x148 bp paired end library. The original data has been mapped to the GRCh37d5 human reference genome using BWA-MEM, and subsequently downsampled to 30x average coverage.

We performed our experiments on Chromosome 20 of the sample, which is a DNA molecule that is 63025520 nucleotides long. We extracted raw unaligned reads for Chromosome 20 from the BAM file into a separate FASTQ file. The size of this FASTQ file is 4.3 GB and it contains 13424120 reads. All of the alignment information was discarded in the transformation from BAM to FASTQ.

The GRCh37 human reference genome was obtained from the 1000 Genomes Project FTP site at ftp://ftp.ncbi.nih.gov/1000genomes/ftp/technical/reference/phase2_reference_assembly_sequence/. The reference is in FASTA format and is 3GB in size. It contains the reference sequence for human chromosomes 1-22, X, Y, MT (mitochondrial) as well as a set of decoy sequences and unplaced

contigs. We use the entire reference genome because reads that were mapped to Chromosome 20 by BWA-MEM may be mapped elsewhere by the minimap2 aligner. We pre-process the reference for use by minimap2 to generate a minimizer index genome.mmi using default parameters.

5.5.3 Germline SNP Calling

For germline SNP calling we put together a pipeline using Rheos services that takes a FASTQ sample file, streams the reads, maps them to a reference genome, processes the variant evidence contained in the reads, save the resulting genomic loci in a data store, and performs variant calling, outputting a result VCF (see Figure 5.44). We then compare the results of the VCF output to the output of other well-established variant callers.

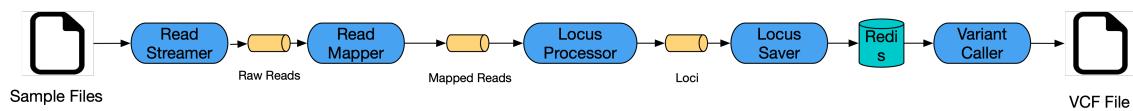


Figure 5.44: The Rheos germline SNP calling pipeline.

In our initial implementation we have actually combined the first two steps, so that the Read Mapping Service reads a FASTQ file directly from disk, rather than a stream, and then streams the already mapped reads for further processing. We use a single instance of the Read Mapping Service for this task invoking it with the following parameters:

```

file -i data/giab/RMNISTHS_30xdownsample_chr20.fastq
-r data/reference/genome.mmi
-g 20:1-63025520
-k kafka:9092
-bin_size=10000
-num_bins=100
-check_freq=1000
-log_level=DEBUG
-root_log_level=DEBUG
-logger_type=stream

```

These parameters indicate that we are processing the entire Chromosome 20 from a FASTQ file, that we would like to use 100 bins of capacity 10000 read pairs per partition in the `PartitionedReservoirSet`, and that we would like to check which bins are full once every 1000 mapped read pairs. Partition size was left at the default 20000000 bases. Using a single CPU and an allotment of 16GB RAM this process took 1 hour and 23 minutes to complete in full, processing 13424120 reads out of which 245052 reads were unpaired, sending 12482 bins for downstream processing. Looking at the system resource usage during the execution (see Figure 5.45) we see that this process appears to be CPU bound, with full utilization over almost the entirety of the processing cycle. Memory usage grows initially as more and more bins accumulate mapped reads, and peaks at 15.5GB as the rate of bin filling is counteracted by sending off bins that are full. Due to the stateless nature of the process, the wall time of the mapping process can be sped up arbitrarily by splitting

the input file into chunks and parallelizing across CPUs.



Figure 5.45: System resource usage by Read Mapping Service during alignment of Chromosome 20 of NA12878 form GIAB.

We next investigate the operation of the Locus Processor Service by simultaneously executing 4 instances of the service, one for each partition of Chromosome 20, on the stream data produced by the Read Mapping Service. We invoke the Locus Processor Service with the following set of parameters:

```
service
-r data/reference/genome.fa
-g [20:1-20000000 / 20:20000001-40000000 / 20:40000001-60000000 / 20:60000001-
63025520]
-k kafka:9092
-log_level=DEBUG
-root_log_level=DEBUG
-logger_type=stream
```

Here, each of the service instances has a separate partition and genomic region assigned via the -g parameter. This process completed in 29 minutes and 42 seconds running four instances simultaneously and took 0.452 seconds to process a single message on average, sending an average of 32908 updated loci for downstream processing per message. Consulting the operational metrics collected during the run (Figure 5.46) we see a fairly stable CPU and memory utilization profiles, however significant use of swap is observed (bottom right panel), indicating that if the amount of memory per process was increased, performance could be improved.



Figure 5.46: System resource usage by Locus Processor Service during processing of Chromosome 20 of NA12878 from GIAB.

We run a single instance of the Locus Saver Service to process locus updates with

the following parameters:

```
service
-k kafka:9092
-r redis-master -p 6379
-log_level=DEBUG
-root_log_level=DEBUG
-logger_type=stream
```

This is a fairly lightweight service and takes 18 minutes and 58 seconds to process all of the messages produced by the 4 instances of Locus Processor and save them to Redis. 961 messages in total are processed, with average processing time of 0.72 seconds per message. Figure 5.47 shows operational metrics collected from the service and the Redis server during the execution runtime. As can be seen, the execution fully utilizes the CPU of the service host (likely for serialization/deserialization of messages). Resources of the Redis server are only mildly utilized indicating the ability to run several instances of the Locus Saver per instance of Redis. The bottom-left panel shows the impact of automated disk persistence of Redis in-memory data. This feature can be turned off to obtain higher throughput at the expense of lower availability, i.e. if Redis crashes data will need to be reprocessed. The last component in the processing pipeline is the Variant Calling Service. We invoke this service with the following parameters:

```
file
-g 20:1-63025520
-r redis-master -p 6379
-calling_threshold=50
-max_vqual=5000
-t variant_caller_pkg/variant_caller/rheos_vcf_template.vcf
-o data/na12878_20_1_63025520_rheos_latest.vcf
-log_level=DEBUG
-root_log_level=DEBUG
-logger_type=stream
```

These parameters indicate that we are calling variants on the entirety of Chromosome 20, that we only include variants exceeding variant quality of 50 (i.e. 1 in 10000 variants is estimated to be a false positive), and with a variant quality ceiling of 5000 (if a variant has vqual >5000 it will be capped at 5000). Running a single instance of this service took 1 hour and 21 minutes to complete calling on the entire Chromosome 20 of the sample, evaluating 12754171 potential variant sites from Redis. Figure 5.48 shows steady CPU utilization and relatively low memory use by the service (top panels), and steady but low CPU utilization, but high memory consumption by Redis (bottom panels), as expected.

Using the procedure above we obtained a germline SNP callset containing 82576 germline variants. We compare the results of our callset to 2 other callsets for the same sample and genomic region:

freebayes - To be able to best compare callsets, we took the same FASTQ file that has been used for the Rheos analysis, and aligned it against the GRCh37 reference genome using minimap2, producing a SAM file. We then sorted and compressed that SAM file using samtools, to produce a BAM file that can be used for variant calling by most tools.



Figure 5.47: System resource usage by Locus Saver Service during processing of Chromosome 20 of NA12878 form GIAB.

We installed the latest version of freebayes[78] and invoked it with the following command:

```
freebayes -r 20 -f reference/genome.fa
giab/RMNISTHS_30xdownsample_chr20.name_sorted_minimap.new_rg.sorted.bam
> tests_latest/original/freebayes_chr20_minimap.vcf
```

After filtering out indels we were left with a callset with 89704 SNPs.

GATK 4 - To obtain a GATK 4 callset. We installed and built the latest version of the GATK[149] from github. Utilizing the same BAM file as was used for the freebayes callset, we invoked the GATK HaplotypeCaller with the following command:

```
java -jar $GATK HaplotypeCaller
-I /data/giab/RMNISTHS_30xdownsample_chr20.name_sorted_minimap.new_rg.sorted.bam
-R /data/reference/genome.fa
-O tests_latest/gatk_chrom20_minimap.vcf
```



Figure 5.48: System resource usage by Variant Caller Service during processing of Chromosome 20 of NA12878 form GIAB.

-pairHMM AVX_LOGLESS_CACHING

-L 20

After removing indels we were left with a callset with 77762 SNPs.

We compared the Rheos callset to those produced by freebayes and GATK by running the Picard[173] tools' GenotypeConcordance utility, with the command:

java -jar \$PICARD GenotypeConcordance

CALL_VCF=tests_latest/original/na12878_20_1_63025520_rheos_latest.vcf

TRUTH_VCF=tests_latest/gatk_chrom20_no_indels.vcf

O=tests_latest/gatk_to_rheos/gatk_to_rheos

TRUTH_SAMPLE=NA12878

CALL_SAMPLE=NA12878 and a similar invocation for the freebayes to Rheos

comparison.

The results were as follows:

Table 5.15: Sensitivity and specificity of Rheos caller vs freebayes and GATK.

Rheos vs.	Sensitivity	Specificity
freebayes	0.988701	0.990385
GATK	0.984848	0.991597

While these results are quite good, and Rheos appears to capture the greatest majority of the signal that other well established tools do on the same data set, it is illuminative to manually consider a sample of loci where the callers disagree. Figure 5.49 shows three examples where variants called by GATK do not appear in the Rheos callset.

In panel a) is a variant that is called a heterozygous SNP by GATK at the locus 930480 on Chromosome 20. Out of 14 reads 4 show the alternate allele T and GATK assigns a variant quality of 64.77. On the Rheos side, the variant was assigned a variant quality of 46.94, and consequently did not pass the quality threshold cutoff of 50 that was used when producing the callset. It is difficult to say based on this data whether this call is likely a true variant or not. Certainly, if a lower stringency cutoff was used the call would also be part of the Rheos callset.

In panel b) is a variant at position 20:2678084 that is called homozygous variant by GATK, but in fact, the coverage at this site is only 2 so GATK is making a call based on very limited information, and the call has a high probability of being a false positive in the GATK callset. Rheos attaches a probability of 0.62 that the site is homozygous variant, and 0.32 that the site is heterozygous variant. The overall variant quality however, is 12 on the account of the low number of observations and consequently the site does not pass the quality threshold.

In panel c) there are two adjacent SNPs in positions 20:18956622-18956623. The first of these is called by both variant callers, but the second is only present in the GATK callset. It is called a heterozygous SNP with 5 out of 19 reads showing the alternate A allele, and has a variant quality of 97.77. In the Rheos callset the variant is assigned a quality of 40.0 and does not pass through the quality cutoff. Because of the lower fraction of alternate allele observations than would be expected, it is possible that the site actually represents a somatic variant that is only present in the subset of the sample's cells.

A unifying theme of all of the false negative calls is that even though they are detected by Rheos, they end up below the quality threshold of 50 (one false variant in 100000). Using a less stringent filter will certainly increase sensitivity of the callset, although an approach for controlling the resultant increase in false positives is also required.

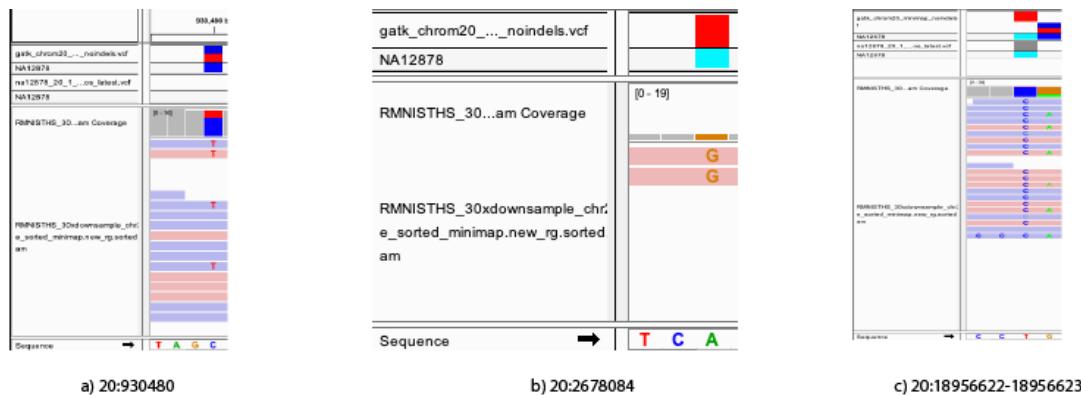


Figure 5.49: The Rehos germline deletion calling pipeline.

Figure 5.50 shows several examples of false positive calls, those that are detected as variant by Rheos but are not called by GATK.

In panel a) we see a locus at 20:813667-813669 that is called as 3 adjacent SNPs by Rheos, whereas in GATK it is called a SNP adjacent to a complex variant. It is indeed probably a complex variant that extends to all 3 adjacent nucleotides, and there is evidence of both the presence of SNPs, as well as a small deletion at that locus. Since Rheos has no concept of indels or complex variants, these sites naturally end up as false positive SNP calls.

In panel b) we see a locus at 20:2487891 that is called a heterozygous SNP by Rheos but is not present in the GATK callset. Rheos calls this site variant because there is a relatively large number (10/24) of alternate allele observations, although they are mostly of relatively low base quality (9 out of 10 reads have base quality between 6 and 8, implying probability of error between 15-25% for each). Additionally, all of the alternative allele observations fall on reads that map to the negative strand. This type of strand bias is a strong indicator of a false positive signal and is often used by variant filtering software to filter out false positives.

In panel c) we see a locus at 20:11298489 that is called homozygous variant in the Rheos callset, but is actually included as part of the neighbouring deletion by GATK. Indeed, this region is called differently by all three callers. GATK calls a 5bp indel at 11298472-11298476, followed by an 8bp indel at 11298482-11298489. Freebayes calls a 13bp indel 11298472-11298484, followed by a homozygous SNP at 11298489. And Rheos calls a bunch of SNPs inside the indel, since it presently has no concept of indels, followed by a homozygous SNP at 11298489.

Two major sources of false positive signals are apparent for Rheos from this testing, areas around indels, and loci with many low quality bases. Both of these warrant further work to improve.



Figure 5.50: Three examples of false negative calls by Rheos, compared to GATK. a) Site with few, but high quality alternate allele observations. b) Site with very low coverage. c) Site of two adjacent SNPs, one of which has low variant quality.

Our testing of Germline SNP Calling with Rheos shows that this framework is capable of deploying scalably onto an academic cloud computing infrastructure and producing accurate results that are compatible with callsets produced by leading variant callers, while doing so in a streamlined, online manner that is not accessible to these other tools.

5.5.4 Germline Deletion Calling

For germline deletion calling we are interested in collecting discordantly mapped read pairs and performing clustering, and variant calling as described in Section 5.3.4. We put together a Rheos pipeline shown in Figure 5.51. We use the same run of the Read Mapping Service as was used for germline SNP calling, but consume all of the reads from all mapped read partitions by the same single instance of the Insert Size Filtering Service.

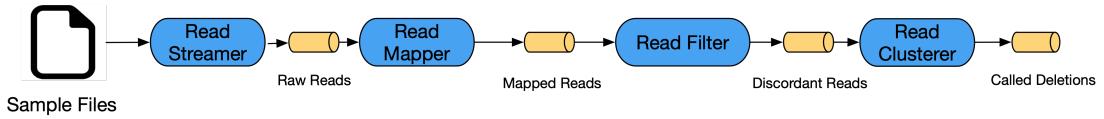


Figure 5.51: Three examples of false positive calls by Rheos, compared to GATK.
 a) Site of a complex variant. b) Site with many low quality bases. c) Site of a SNP adjacent to an indel.

We use the following settings for the Insert Size Filtering Service - accumulate 100000 read pairs for insert size estimation, keep read pairs with insert size $> 5 * \text{MAD}$ (Median Absolute Deviation), that are both mapped, that map to the same contig, and that both have mapping quality ≥ 30 . These settings result in 2663 read pairs passing through the Insert Size Filtering Service to the Insert Size Clustering Service. The total runtime was 4 minutes 32 seconds to process 13069 messages from 168 data partitions.

We use a single instance of the Insert Size Clustering Service applying the following settings - $\text{min_read_support} = 2$, $\text{insert_size_threshold} = 500$, indicating that we only want deletions that are supported by at least two discordant reads, and that we want to remove from a deletion's list of supporting reads, those that have $\text{insert_size} > \text{deletion_width} + 500$. We apply KDE clustering with a gaussian kernel, setting $\text{bandwidth} = 100$. The runtime of the clustering and deletion calling was 17 seconds. Results of the clustering can be seen in Figure 5.23.

The initial calling produced a set of 36 putative deletions. Looking at the read pairs that support these deletions we observed that a deletion was supported, on average by 121 read pairs. This is a very high number given that the total number of read pairs considered is 2663. This observation has led to the consideration of whether there is a significant number of potentially spurious read pairs with very long insert sizes that overlap many deletions, but do not actually contribute to the signal. We investigated the distribution of insert size widths in the data set, with results visible in Figure 5.52. It is evident that even though the majority of insert sizes are relatively small - $< e^{10}$, there are over a hundred read pairs with insert size $> e^{14}$.

We would like to investigate whether the large insert-size read pairs contribute significantly to signal identified by one-dimensional KDE, and indeed, whether they may cluster by themselves. To assess this, we introduce a second dimension of insert size to the original clustering, and perform a second round of KDE clustering in two dimensions, where each point was now of the form $(\text{center}, \text{width})$. The results can be seen in Figure 5.53. Here, the horizontal dimensions are insert size center and width, and the vertical dimension is density. We can see that most of the density still falls in the area of low width and along centre coordinate. We thus, believe that the very long insert sizes are not significant contributors to the detection of deletions that they overlap, and may be removed from the list of reads supporting those deletions. Indeed, when we remove these read pairs from our data set entirely and repeat the clustering, the results recapitulate those produced with the original data set, and only one fewer deletions is called. We thus introduce a width-based cutoff for reads supporting a deletion and set it to $\text{deletion_width} + 500$, where all read pairs that have insert size greater than this cutoff are removed from the list of supporting reads. This results in a much more sensible average number of support-

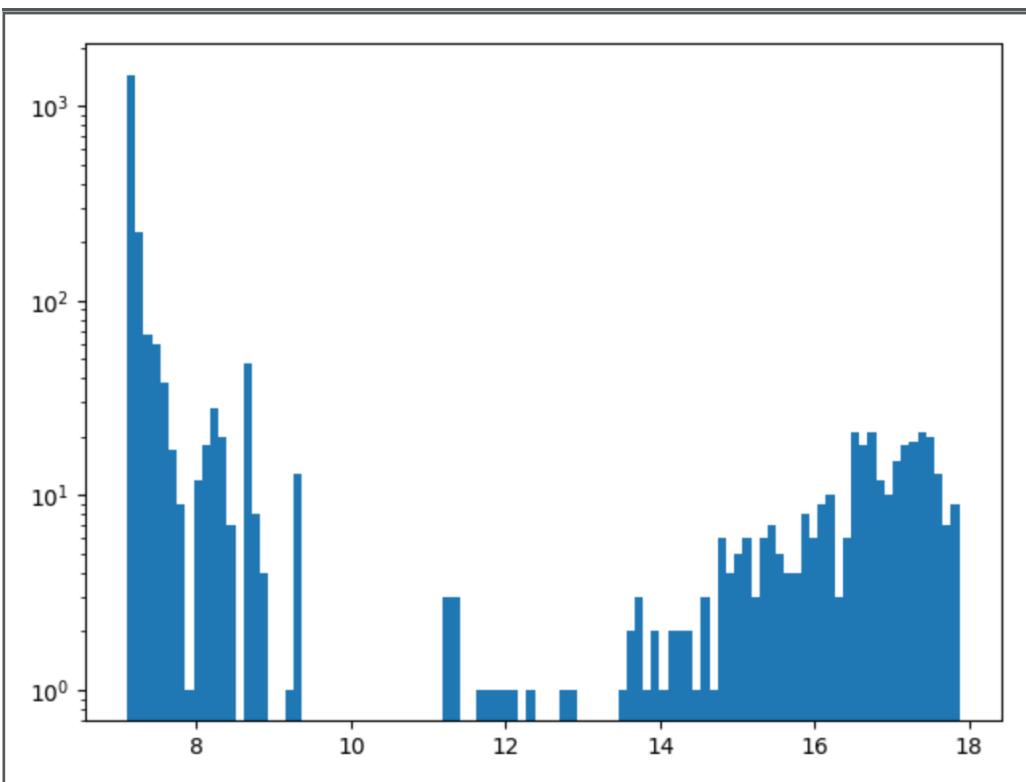


Figure 5.52: A histogram of discordant read pair insert sizes on the log-log scale. X-axis is $\log(\text{insert_size})$, Y-axis is $\log(\text{count})$.

ing reads per deletion, namely 9. We retain those deletions for our final callset that are supported by at least 2 read pairs, leaving 29 deletions in the final callset.

We compare the results of this method with two other data sets. One is a set of structural variant calls produced by the GIAB consortium. We downloaded the file Personalis_1000_Genomes_deduplicated_deletions from ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/technical/svclassify_Manuscript/Supplementary_Information/. We then used the software package *bedtools*[175] to filter the data set to only include deletions on Chromosome 20 and those exceeding 500bp in length. The length trimming is necessary because the insert size based method of Rheos is limited to detecting deletions that are approximately equal to the $MAD * \text{mad_threshold}$ in size. MAD is 131.95 in our data set and *mad_threshold* is 5. We take the cutoff at 500 since slightly smaller deletions can be detected, due to the overlap of multiple read pairs. This results in a callset of 26 deletions.

The other comparison data set is produced by calling variants with *Delly*[177]. We produce the callset by invoking *Delly* with the following command:

```
delly call -n -q 30 -s 5
-g reference/genome.fa giab/RMNISTHS_30xdownsample_chr20.minimap.sorted.bam
-o tests_latest/delly_chrom20_dels_minimap.vcf
```

The flags supplied to *Delly* have been selected to mimic the cutoffs used for the Rheos caller. We then trim the callset to only include deletions that are not flagged as *LowQuality*, and that are greater than 500 bp in length. This results in a callset of 25 deletions.

We used *bedtools* to compare the 3 callsets and produce the diagram of Figure 5.54

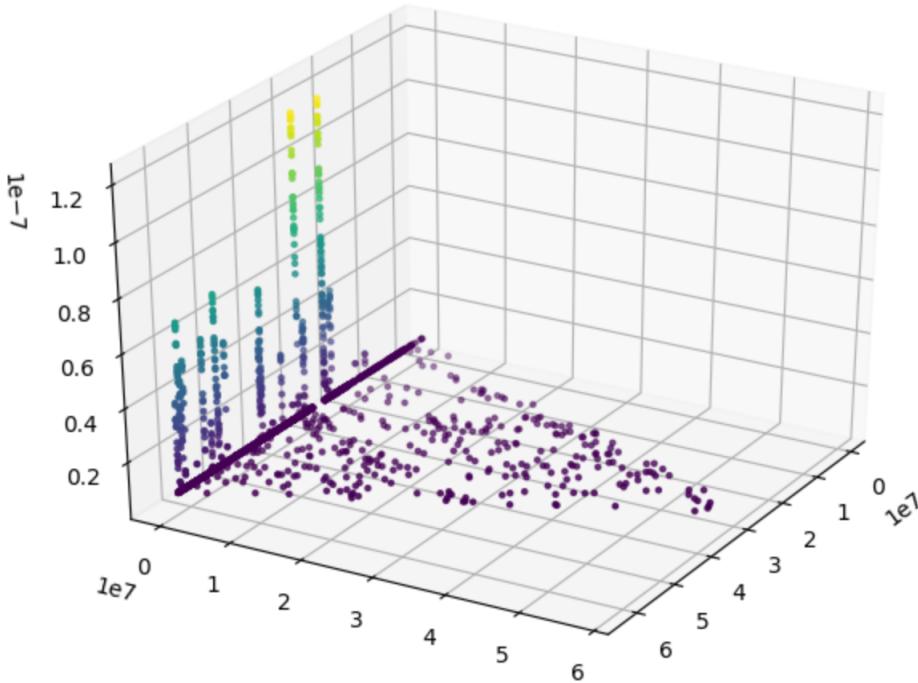


Figure 5.53: 2-D KDE estimate of insert size center and width, using a gaussian kernel with bandwidth 100.

that shows the overlap between them. 83% of the deletions called by Rheos are called in at least one of the other two callsets. This is a pretty satisfactory result as the Rheos callset only makes use of discordantly mapped reads at this time, while the other two callsets have been produced with methods that take advantage of split-reads and read depth analysis.

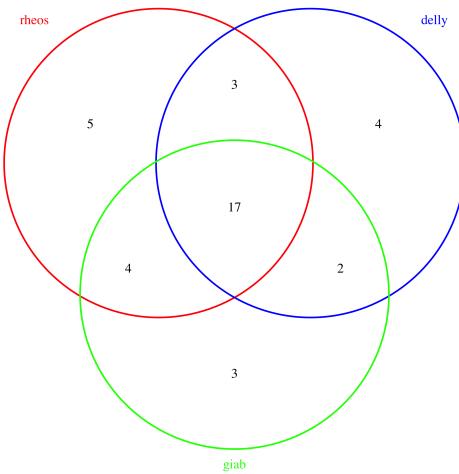


Figure 5.54: Venn diagram depicting overlap between the Rheos, Delly, and GIAB deletion callsets on Chromosome 20 of NA12878.

To assess the shortcomings of the Rheos deletion calling method we take a look at a sample of the deletions that are called by Rheos and not by other tools, and also those that are called by other tools but not by Rheos. Figure 5.55 shows the

5 deletion calls that were called only by Rheos. Out of these five calls, three (a), b), and e)) look like real deletions when the region is viewed with IGV. There is a clear depletion of reads in these regions compared to the surrounding regions and there are several discordantly mapped read pairs that overlay each region (3 each for deletions a), b), and c)). The region in panel b) is difficult to make calls in, because of the large number of low quality reads, with large portions clipped out of the alignment and the call is only supported by two read pairs so this can easily be a false positive. The region in d) is hard to judge, but can also be a false positive.

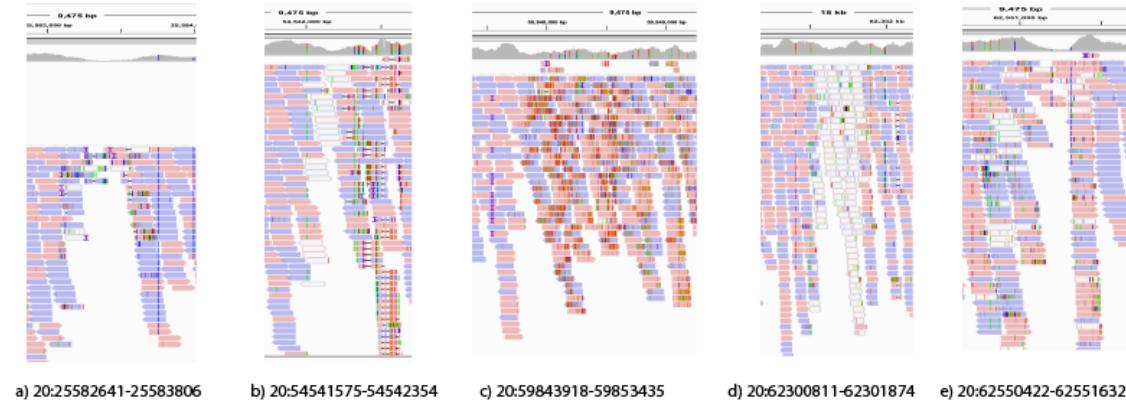


Figure 5.55: Visualization of the genomic regions that were called deletions by Rheos but not by other callers.

In Figure 5.56 we show several examples of deletion calls that were not part of Rheos' callset but were called by other tool. The deletions in panels a) and b) were part of the GIAB callset and part of the Delly callset. These look like very well defined deletions with a clear dropoff in read coverage, as well as split-reads flanking the deletion site. Since Rheos is currently not using read depth or split-reads, it was not able to make these calls. When looking at the deletion call in panel c) is not readily apparent that this is a deletion. Even though there are some split-reads at the flank, there is no dropoff in coverage. It may be that this is a false positive call in the GIAB callset. The deletion in panel d) was called by Delly, but is not in GIAB and was not called by Rheos. This region is difficult to judge by sight. There is clearly a small homzygous deletion in the left portion of the image, but the Delly deletion call spans the entire visible region. There is a clear difference in read depth visible, but only on one side of the region, the read depth towards the left is relatively constant. The only true variant in this region is likely the small deletion, but since it is too small for Delly to call, it is probably extending the call towards the right, thus creating a false variant.

Based on the manual inspection of called regions that we performed, it appears that the true sensitivity and specificity of the Rheos deletion calling method would be underestimated by a direct comparison with the two other callsets, because they miss some calls that appear to be true, and because they make some calls that appear to be false. Overall, the currently implemented method of germline deletion calling in Rheos appears to be accurate, but will certainly benefit from the additional signal sources like split-reads and read depth that other tools make use of.

In our experimental validation work we were successfully able to deploy Rheos on cloud computing infrastructure using Google Kubernetes. We executed several real end-to-end variant calling use cases using a data set of 4.7 GB in size. We success-

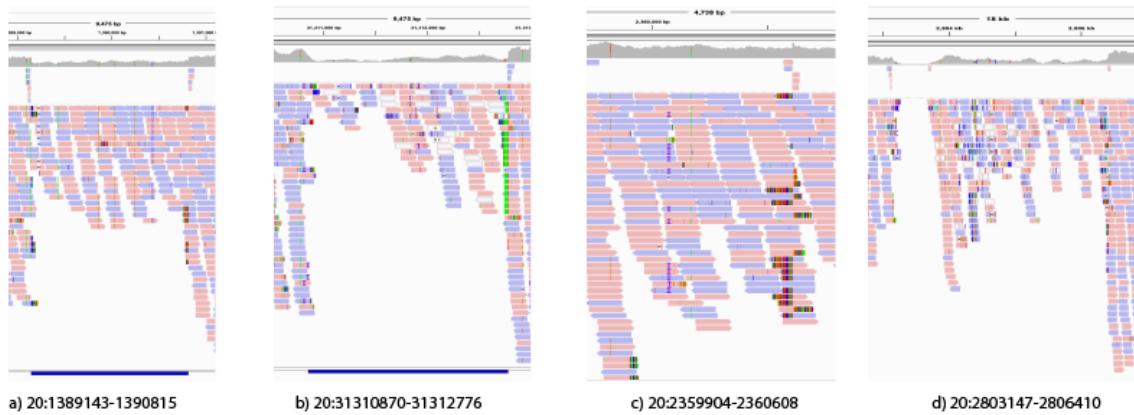


Figure 5.56: Visualization of the genomic regions that were called deletions by both GIAB and Delly but not Rheos (a) and b)), GIAB but not Rheos or Delly (c)), Delly but not GIAB or Rheos (d)).

fully compared the results to the data produced by other well established variant callers and found them to be accurate. Future enhancements of the Rheos framework will surely improve both the scalability of the framework, as well as the accuracy of the callsets that it produces.

Chapter 6

Discussion and Conclusion

6.1 Validation and Conclusion

We have deployed Butler in a production setting at the EMBL/EBI's Embassy Cloud in a configuration that utilizes 1500 CPUs, 6 TB RAM, 1 PB of Isilon storage accessed over NFS, and 40 TB of block-storage. Furthermore, we have built a series of workflows that facilitate the large-scale cancer genomics analyses carried out by the Germline Working Group of the Pan Cancer Analysis of Whole Genomes project, including:

- Germline SNV discovery
- Germline SNV joint-genotyping
- Germline SV genotyping
- Variant Filtration
- Sample submission

Using these workflows we have carried out a number of analyses on a 725TB data set of 2834 cancer patients' DNA samples consuming a total of 546,552 CPU hours. Each analysis took no longer than two weeks to complete and utilized only 1.5% - 2.2% of the overall compute capacity for management overhead. On several occasions we were able to detect large scale cluster instability and program crashes utilizing the Operational Management system and take corrective action with a minimal impact on overall cluster productivity.

Subsequent to the success of these analyses several research groups from the European Bioinformatics Institute, Ontario Institute for Cancer Research, Francis Crick Institute, and the Centre for Genomic Regulation have expressed their interest in utilizing Butler for their own large scale analyses in the cloud.

Based on the adherence of the Butler design and implementation to the stated set of requirements, and sustained successful production operation in a large scale deployment on a multitude of scientific analyses of significant scope and size, we conclude that the Butler framework is an effective tool for large scale scientific workflow management in the cloud.

6.2 Future Direction

Butler has been created to facilitate scientific analyses at scale and we have demonstrated that it is able to successfully perform at the level required for today’s big data initiatives in the genomics domain. There are projects on the horizon, however, that are one to two orders of magnitude larger than the current biggest projects, these include the UK’s 100,000 Genomes Project[148], and the US Precision Medicine Initiative[35] (with up to 1,000,000 genomes). This means that in order to not have to proportionately increase the timeline for these projects the computational infrastructure will have to be scaled up instead. It is thus imperative for Butler’s continued relevance to be able to ascertain the framework’s performance level at 1 or 2 orders of magnitude larger than the current 1500 core empirically obtained result. The most immediate opportunity to do so will come up in 2017 when the EMBL/EBI’s Embassy Cloud will be upgraded to 5000 CPU cores and Butler has been invited to take part in the stress-testing of the upgraded cloud.

It is important to grow the library of workflows that are readily available for the Butler system to make the framework more appealing to new users. The Technical Working Group of the PCAWG project is in the process of migrating all of the main computational pipelines that have been used in the project into Docker[152] containers. Although the workflows that have been developed for the Germline Working Group have not yet been ported to Docker, Airflow, the workflow system underlying Butler has support for running Docker containers. Thus, a key next step for growing the library of Butler workflows lies in the adaptation of the core PCAWG workflows to be able to easily run them on a Butler instance. This would allow Butler to offer a comprehensive set of next generation sequencing workflows that are used for cancer genomics analysis.

Deploying Butler to a larger variety of environments will confirm the multi-cloud purpose of the framework and allow for the development of a richer set of configuration and provisioning profiles, as necessitated by the differences between deployment environments. On the basis of the already completed analyses for the PCAWG Germline Working Group, the Butler framework has also been selected to help deliver the science demonstrator work packet of the European Open Science Cloud Pilot[63] initiative that is launching in 2017. Additionally, de.NBI - The German Network for Bioinformatics Infrastructure[95] which is working to establish a German academic cloud computing environment for bioinformatics research will be using Butler to deliver a number of new bioinformatics pipelines on its cloud in 2017.

Thus, over the course of the next 12 months the focus of Butler development will be on supporting improved scalability, developing a richer set of computational pipelines and operating in a number of new cloud computing environments. These steps should result in a more robust, feature rich, and useful tool.

Appendix A

Code Listings

Listing 14: Terraform configuration of a worker VM

```
1 provider "openstack" {
2     user_name = "${var.user_name}"
3     password = "${var.password}"
4     tenant_name = "${var.tenant_name}"
5     auth_url = "${var.auth_url}"
6 }
7
8 resource "openstack_compute_instance_v2" "worker" {
9     image_id = "${var.image_id}"
10    flavor_name = "s1.massive"
11    security_groups = ["internal"]
12    name = "${concat("worker-", count.index)}"
13    network = {
14        uuid = "${var.main_network_id}"
15    }
16    connection {
17        user = "${var.user}"
18        key_file = "${var.key_file}"
19        bastion_key_file = "${var.bastion_key_file}"
20        bastion_host = "${var.bastion_host}"
21        bastion_user = "${var.bastion_user}"
22        agent = "true"
23    }
24    count = "175"
25    key_pair = "${var.key_pair}"
26    provisioner "remote-exec" {
27        inline = [
28            "sudo mv /home/centos/saltstack.repo
29             /etc/yum.repos.d/saltstack.repo",
30            "sudo yum install salt-minion -y",
31            "sudo service salt-minion stop",
```

```

32         "echo 'master: ${var.salt_master_ip}' | sudo tee -a
33             ↵ /etc/salt/minion",
34         "echo 'id: ${concat("worker-", count.index)}' | sudo tee
35             ↵ -a /etc/salt/minion",
36         "echo 'roles: [worker, germline, consul-client]' | sudo
37             ↵ tee -a /etc/salt/grains",
38         "sudo hostname ${concat("worker-", count.index)}",
39         "sudo service salt-minion start"
40     ]
41 }

```

Listing 15: Terraform configuration of a security group

```

1 resource "openstack_compute_secgroup_v2" "internal" {
2     name = "internal"
3     description = "Allows communication between instances"
4     #SSH
5     rule {
6         from_port = 22
7         to_port = 22
8         ip_protocol = "tcp"
9         self = "true"
10    }
11    #Saltstack
12    rule {
13        from_port = 4505
14        to_port = 4506
15        ip_protocol = "tcp"
16        self = "true"
17    }
18 }

```

Listing 16: Salt Pillar for specifying test data location.

```

1 test_data_sample_path: /shared/data/samples
2
3 test_data_base_url: http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/
4
5 test_samples:
6   NA12874:
7   -
8     - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam
9     - 88a7a346f0db1d3c14e0a300523d0243
10    -

```

```
11      - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam.bai
12      - e61c0668bbaacdea2c66833f9e312bbb
```

Listing 17: Using Salt Mine to look up a server's IP Address.

```
1 consul-client:
2     service.running:
3         - enable: True
4         - watch:
5             - file: /etc/opt/consul.d/*
6 {%- set servers = salt['mine.get']('roles:(consul-server|consul-bootstrap)', 
7     'network.ip_addrs', 'grain_pcre').values() %}
8 {%- set node_ip = salt['grains.get']('ip4_interfaces')['eth0'] %}
9 # Create a list of servers that can be used to join the cluster
10 {%- set join_server = [] %}
11 {%- for server in servers if server[0] != node_ip %}
12     {%- do join_server.append(server[0]) %}
13 {%- endfor %}
14 join-cluster:
15     cmd.run:
16         - name: consul join {{ join_server[0] }}
17         - watch:
18             - service: consul-client
```

Listing 18: Using Top File to map States to Roles.

```
1 base:
2     '*':
3         - consul
4         - dnsmasq
5         - collectd
6     'G@roles:monitoring-server':
7         - influxdb
8         - grafana
9     'G@roles:job-queue':
10        - rabbitmq
```

Listing 19: Collectd configuration for metrics collection.

```
1 # Read metrics about cpu usage
2 [[inputs.cpu]]
3     ## Whether to report per-cpu stats or not
4     percpu = true
5     ## Whether to report total system cpu stats or not
```

```

6      totalcpu = true
7      ## If true, collect raw CPU time metrics.
8      collect_cpu_time = false
9      ## If true, compute and report the sum of all non-idle CPU states.
10     report_active = false
11
12
13 # Read metrics about disk usage by mount point
14 [[inputs.disk]]
15     ## By default, telegraf gather stats for all mountpoints.
16     ## Setting mountpoints will restrict the stats to the specified mountpoints.
17     # mount_points = ["/"]
18
19     ## Ignore some mountpoints by filesystem type. For example (dev)tmpfs (usually
20     ## present on /run, /var/run, /dev/shm or /dev).
21     ignore_fs = ["tmpfs", "devtmpfs", "devfs"]
22
23
24 # Read metrics about disk IO by device
25 [[inputs.diskio]]
26     ## By default, telegraf will gather stats for all devices including
27     ## disk partitions.
28     ## Setting devices will restrict the stats to the specified devices.
29     # devices = ["sda", "sdb"]
30     ## Uncomment the following line if you need disk serial numbers.
31     # skip_serial_number = false
32     #
33     ## On systems which support it, device metadata can be added in the form of
34     ## tags.
35     ## Currently only Linux is supported via udev properties. You can view
36     ## available properties for a device by running:
37     ## 'udevadm info -q property -n /dev/sda'
38     # device_tags = ["ID_FS_TYPE", "ID_FS_USAGE"]
39     #
40     ## Using the same metadata source as device_tags, you can also customize the
41     ## name of the device via templates.
42     ## The 'name_templates' parameter is a list of templates to try and apply to
43     ## the device. The template may contain variables in the form of '$PROPERTY' or
44     ## '${PROPERTY}'. The first template which does not contain any variables not
45     ## present for the device is used as the device name tag.
46     ## The typical use case is for LVM volumes, to get the VG/LV name instead of
47     ## the near-meaningless DM-0 name.
48     # name_templates = ["$ID_FS_LABEL", "$DM_VG_NAME/$DM_LV_NAME"]
49
50
51 # Get kernel statistics from /proc/stat
52 [[inputs.kernel]]
53     # no configuration

```

```
54
55
56 # Read metrics about memory usage
57 [[inputs.mem]]
58     # no configuration
59
60
61 # Get the number of processes and group them by status
62 [[inputs.processes]]
63     # no configuration
64
65
66 # Read metrics about swap memory usage
67 [[inputs.swap]]
68     # no configuration
69
70
71 # Read metrics about system load & uptime
72 [[inputs.system]]
73     # no configuration
```

Listing 20: Filebeat Prospector configuration.

```
1 [collectd]
2     enabled = true
3     bind-address = ":8096"
4     database = "metrics"
5     retention-policy = ""
6     batch-size = 5000
7     batch-pending = 10
8     batch-timeout = "10s"
9     read-buffer = 0
10    typesdb = "/usr/share/collectd/types.db"
```

Listing 21: TICKscript for alerting on CPU value.

```
1 // Parameters
2 var info = 70
3 var warn = 80
4 var crit = 90
5 var infoSig = 2.5
6 var warnSig = 3
7 var critSig = 3.5
8 var period = 10s
9 var every = 10s
```

```

10
11 // Dataframe
12 var data = stream
13 |from()
14   .database('metrics')
15   .retentionPolicy('default')
16   .measurement('cpu_value')
17   .where(lambda: "type" == 'percent' AND "type_instance" == 'idle')
18 |eval(lambda: 100 - "value")
19   .as('used')
20 |window()
21   .period(period)
22   .every(every)
23 |mean('used')
24   .as('stat')
25
26 // Thresholds
27 var alert = data
28 |eval(lambda: sigma("stat"))
29   .as('sigma')
30   .keep()
31 |alert()
32   .id('{{ index .Tags "host"}}/cpu_value')
33   .message('{{ .ID }}:{{ index .Fields "stat" }}')
34   .info(lambda: "stat" > info OR "sigma" > infoSig)
35   .warn(lambda: "stat" > warn OR "sigma" > warnSig)
36   .crit(lambda: "stat" > crit OR "sigma" > critSig)
37
38 // Alert
39 alert
40   .log('/tmp/cpu_alert_log_2.txt')

```

Listing 22: TICKscript for handling dead VMs.

```

1 {% raw %}
2 var db = 'telegraf'
3 var rp = 'autogen'
4 var measurement = 'system'
5 var groupBy = ['host']
6 var whereFilter = lambda: TRUE
7 var period = 30s
8 var name = 'Host Deadman'
9 var idVar = name + ':{{.Group}}'
10 var blah = '{{index .Tags "host"}}'
11 var message = 'The host {{index .Tags "host"}} is offline as of {{.Time}}.'
12 var messageN = 'The host {{index .Tags "host"}} is back online at {{.Time}}.'
13 var idTag = 'alertID'

```

```
14 var levelTag = 'level'
15 var messageField = 'message'
16 var durationField = 'duration'
17 var outputDB = 'chronograf'
18 var outputRP = 'autogen'
19 var outputMeasurement = 'alerts'
20 var triggerType = 'deadman'
21 var threshold = 0.0
22 var data = stream
23     |from()
24         .database(db)
25         .retentionPolicy(rp)
26         .measurement(measurement)
27         .groupBy(groupBy)
28         .where(whereFilter)
29
30 var trigger = data
31     |deadman(threshold, period)
32         .stateChangesOnly()
33         .message('{{ if eq .Level "CRITICAL" }}' + message + '{{else}}' +
34             ↵ messageN + '{{end}}')
35         .id(idVar)
36         .idTag(idTag)
37         .levelTag(levelTag)
38         .messageField(messageField)
39         .durationField(durationField)
40         .slack()
41         .channel('#embassyalerts')
42         {% endraw %}
43         .exec('butler_healing_agent', 'relaunch-worker', '-t', '{{
44             ↵ pillar['terraform_files']] }}', '-s', '{{ pillar['terraform_state']
45             ↵ }}', '-v', '{{ pillar['terraform_vars']] }}', '-p', '{{
46             ↵ pillar['terraform_provider']] }}')
47
48 trigger
49     |eval(lambda: "emitted")
50         .as('value')
51         .keep('value', messageField, durationField)
52     |influxDBOut()
53         .create()
54         .database(outputDB)
55         .retentionPolicy(outputRP)
56         .measurement(outputMeasurement)
57         .tag('alertName', name)
58         .tag('triggerType', triggerType)
59
60 trigger
61     |httpOut('output')
```

58 {%- endraw %}

Listing 23: Butler healing agent code for restarting the Airflow Scheduler.

```

1  def call_command(command, cwd=None):
2      try:
3          logging.debug("About to invoke command: " + command)
4          my_output = check_output(command, shell=True, cwd=cwd, stderr=STDOUT)
5          logging.debug("Command output is: " + my_output)
6          return my_output
7      except CalledProcessError as e:
8          logging.error("An error occurred! Command output is: " +
9                         e.output.decode("utf-8"))
10         raise
11
12 def is_critical(level):
13     return level == "CRITICAL"
14
15 def parse_alert_data():
16     return json.loads(sys.stdin.read())
17
18 def get_host_name(alert_data):
19     return alert_data["data"]["series"][0]["tags"]["host"]
20
21 def restart_service(host, service_name):
22     call_command("pepper {} service.restart {}".format(host, service_name), None)
23
24 def parse_args():
25     my_parser = argparse.ArgumentParser()
26
27     sub_parsers = my_parser.add_subparsers()
28
29     common_args_parser = argparse.ArgumentParser(
30         add_help=False, conflict_handler='resolve')
31
32     restart_airflow_scheduler_parser = sub_parsers.add_parser(
33         "restart-airflow-scheduler", parents=[common_args_parser],
34         conflict_handler='resolve')
35
36     restart_airflow_scheduler_parser.set_defaults(func=restart_airflow_scheduler_command)
37
38     def restart_airflow_scheduler_command(args, alert_data):
39         if is_critical(alert_data["level"]):
40             restart_service("-G 'roles:tracker'", "airflow-scheduler")

```

Listing 24: Butler healing agent code for relaunching a failed VM.

```
1 def call_command(command, cwd=None):
2     try:
3         logging.debug("About to invoke command: " + command)
4         my_output = check_output(command, shell=True, cwd=cwd, stderr=STDOUT)
5         logging.debug("Command output is: " + my_output)
6         return my_output
7     except CalledProcessError as e:
8         logging.error("An error occurred! Command output is: " +
9             e.output.decode("utf-8"))
10    raise
11
12 def is_critical(level):
13     return level == "CRITICAL"
14
15 def parse_alert_data():
16     return json.loads(sys.stdin.read())
17
18 def get_host_name(alert_data):
19     return alert_data["data"]["series"][0]["tags"]["host"]
20
21 def restart_service(host, service_name):
22     call_command("pepper {} service.restart {}".format(host, service_name), None)
23
24 def parse_args():
25     my_parser = argparse.ArgumentParser()
26
27     sub_parsers = my_parser.add_subparsers()
28
29     common_args_parser = argparse.ArgumentParser(
30         add_help=False, conflict_handler='resolve')
31
32     relaunch_worker_parser = sub_parsers.add_parser(
33         "relaunch-worker", parents=[common_args_parser],
34         conflict_handler='resolve')
35     relaunch_worker_parser.add_argument(
36         "-t", "--terraform_location", help="Location of the terraform definition",
37         files.,
38         dest="terraform_location", required=True)
39     relaunch_worker_parser.add_argument(
40         "-s", "--terraform_state_location", help="Location of the terraform state",
41         file.,
42         dest="terraform_state_location", required=True)
43     relaunch_worker_parser.add_argument(
44         "-v", "--terraform_var_file_location", help="Location of the terraform vars",
45         file.,
46         dest="terraform_var_file_location", required=True)
47     relaunch_worker_parser.add_argument(
48         "-p", "--terraform_provider", help="The terraform provider to use.",
```

```

44     choices = provider_list,
45     dest="terraform_provider", required=True)
46 relaunch_worker_parser.set_defaults(func=relaunch_worker_command)
47
48 def is_key_present(key_data, host_name):
49     parsed_key_data = json.loads(key_data)
50     return_data = parsed_key_data["return"][0]["data"]["return"]
51
52     if "minions" in return_data:
53         return_vals = return_data["minions"]
54         for val in return_vals:
55             if val == host_name:
56                 return True
57
58     return False
59
60 def locate_minion_key(host_name):
61     minion_connect_try = 1
62     while minion_connect_try <= MINION_CONNECT_MAX_RETRIES:
63         logging.info("Attempt #{} of {} to retrieve minion key for host {} from the
64             ↵ master.".format(minion_connect_try, MINION_CONNECT_MAX_RETRIES,
65             ↵ host_name))
66         key_data = call_command("pepper --client=wheel key.name_match
67             ↵ match={}".format(host_name))
68         logging.debug("Retrieved key data: " + key_data)
69         if is_key_present(key_data, host_name):
70             return True
71         else:
72             logging.debug("Key data for host {} not found at time {}. Sleeping for
73                 ↵ {} seconds.".format(host_name, datetime.now(),
74                 ↵ MINION_CONNECT_SLEEP_PERIOD))
75             time.sleep(MINION_CONNECT_SLEEP_PERIOD)
76             minion_connect_try = minion_connect_try + 1
77
78
79     return False
80
81
82 def relaunch_worker_command(args, alert_data):
83     if is_critical(alert_data["level"]):
84         host_name = get_host_name(alert_data)
85
86         tf_location = args.terraform_location
87         tf_state_location = args.terraform_state_location
88         tf_var_file_location = args.terraform_var_file_location
89         tf_resource = provider_resource_lookup[args.terraform_provider]
90         worker_number = host_name.split("-")[1]
91
92         call_command("pepper --client=wheel key.delete match={}".format(host_name))

```

```
87     call_command("terraform taint -lock=false -state={}
88     ↪  {}.worker.{}".format(tf_state_location, tf_resource, worker_number),
89     ↪  tf_location)
90
91     call_command("terraform apply -lock=false -state={} --var-file {}
92     ↪  -auto-approve".format(tf_state_location, tf_var_file_location),
93     ↪  tf_location)
94
95     locate_minon_key(host_name)
```

Listing 25: Consul service definition for PostgreSQL.

```
1  {
2      "results_base_path": "/shared/data/results/discovery/",
3      "results_local_path": "/tmp/discovery/",
4      "freebayes": {
5          "mode": "discovery",
6          "flags": "--min-repeat-entropy 1
7              ↪  --report-genotype-likelihood-max"
8      }
9 }
```

Listing 26: Source code for the freebayes workflow.

```
1  from airflow import DAG
2  from airflow.operators import BashOperator, PythonOperator
3  from datetime import datetime, timedelta
4
5  import os
6  import logging
7  from subprocess import call
8
9  import tracker.model
10 from tracker.model.analysis_run import *
11 from tracker.util.workflow_common import *
12
13
14 def run_freebayes(**kwargs):
15
16     config = get_config(kwargs)
17     logger.debug("Config - {}".format(config))
```

```

18
19     sample = get_sample(kwargs)
20
21     contig_name = kwargs["contig_name"]
22     contig_whitelist = config.get("contig_whitelist")
23
24
25     if not contig_whitelist or contig_name in contig_whitelist:
26
27         sample_id = sample["sample_id"]
28         sample_location = sample["sample_location"]
29
30         result_path_prefix = config["results_local_path"] + "/" + sample_id
31
32         if (not os.path.isdir(result_path_prefix)):
33             logger.info(
34                 "Results directory {} not present,
35                 ↳ creating.".format(result_path_prefix))
36             os.makedirs(result_path_prefix)
37
38         result_filename = "{}_{}_{}.vcf".format(
39             result_path_prefix, sample_id, contig_name)
40
41         freebayes_path = config["freebayes"]["path"]
42         freebayes_mode = config["freebayes"]["mode"]
43         freebayes_flags = config["freebayes"]["flags"]
44
45         reference_location = config["reference_location"]
46
47         if freebayes_flags == None:
48             freebayes_flags = ""
49
50         if freebayes_mode == "discovery":
51             freebayes_command = "{} -r {} -f {} {} {} > {}".\
52                             format(freebayes_path,
53                                 contig_name,
54                                 reference_location,
55                                 freebayes_flags,
56                                 sample_location,
57                                 result_filename)
58         elif freebayes_mode == "regenotyping":
59             variants_location = config["variants_location"]
60
61             freebayes_command = "{} -r {} -f {} -o {} {} {} > {}".\
62                             format(freebayes_path,
63                                 contig_name,
64                                 reference_location,
65                                 variants_location[contig_name],

```

```
65             freebayes_flags,
66             sample_location,
67             result_filename)
68     else:
69         raise ValueError("Unknown or missing freebayes_mode -
70                           {}".format(freebayes_mode))
71
72     call_command(freebayes_command, "freebayes")
73
74     compressed_sample_filename = compress_sample(result_filename, config)
75     generate_tabix(compressed_sample_filename, config)
76     copy_result(compressed_sample_filename, sample_id, config)
77 else:
78     logger.info(
79         "Contig {} is not in the contig whitelist,
80         skipping.".format(contig_name))
81
82 default_args = {
83     'owner': 'airflow',
84     'depends_on_past': False,
85     'start_date': datetime.datetime(2020, 01, 01),
86     'email': ['airflow@airflow.com'],
87     'email_on_failure': False,
88     'email_on_retry': False,
89     'retries': 1,
90     'retry_delay': timedelta(minutes=5),
91 }
92
93 dag = DAG("freebayes", default_args=default_args,
94            schedule_interval=None, concurrency=10000, max_active_runs=2000)
95
96 start_analysis_run_task = PythonOperator(
97     task_id="start_analysis_run",
98     python_callable=start_analysis_run,
99     provide_context=True,
100    dag=dag)
101
102 validate_sample_task = PythonOperator(
103     task_id="validate_sample",
104     python_callable=validate_sample,
105     provide_context=True,
106     dag=dag)
107
108 validate_sample_task.set_upstream(start_analysis_run_task)
109
110
```

```

111 complete_analysis_run_task = PythonOperator(
112     task_id="complete_analysis_run",
113     python_callable=complete_analysis_run,
114     provide_context=True,
115     dag=dag)
116
117 for contig_name in tracker.util.workflow_common.CONTIG_NAMES:
118     freebayes_task = PythonOperator(
119         task_id="freebayes_" + contig_name,
120         python_callable=run_freebayes,
121         op_kwargs={"contig_name": contig_name},
122         provide_context=True,
123         dag=dag)
124
125     freebayes_task.set_upstream(validate_sample_task)
126
127     complete_analysis_run_task.set_upstream(freebayes_task)

```

Listing 27: Saltstack state for workflow deployment.

```

1 current_runs = run_session.query(Configuration.config[("sample", "
2     ↳ sample_id")]).astext).\
3         join(AnalysisRun, AnalysisRun.config_id == Configuration.config_id).\
4         join(Analysis, Analysis.analysis_id == AnalysisRun.analysis_id).\
5         filter(and_(Analysis.analysis_id == analysis_id, AnalysisRun.run_status
6             ↳ != tracker.model.analysis_run.RUN_STATUS_ERROR)).all()
7
8 available_samples = sample_session.query(PCAWGSample.index.label("index"),
9     ↳ sample_id.label("sample_id"), sample_location.label("sample_location")).\
10        join(SampleLocation, PCAWGSample.index == SampleLocation.donor_index).\
11        filter(and_(sample_location != None, sample_id.notin_(current_runs))).\
12        limit(num_runs).all()

```

Listing 28: Butler Analysis configuration for SNP genotyping.

```

1 {
2     "variants_location": {
3         "1": "/freebayes.chr_1.sites.snv_indel.annot.final.vcf.gz",
4         "2": "/freebayes.chr_2.sites.snv_indel.annot.final.vcf.gz",
5         "3": "/freebayes.chr_3.sites.snv_indel.annot.final.vcf.gz",
6         "4": "/freebayes.chr_4.sites.snv_indel.annot.final.vcf.gz",
7         "5": "/freebayes.chr_5.sites.snv_indel.annot.final.vcf.gz",
8         "6": "/freebayes.chr_6.sites.snv_indel.annot.final.vcf.gz",
9         "7": "/freebayes.chr_7.sites.snv_indel.annot.final.vcf.gz",
10        "8": "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",

```

```
11         "9": "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
12         "10": "/freebayes.chr_10.sites.snv_indel.annot.final.vcf.gz",
13         "11": "/freebayes.chr_11.sites.snv_indel.annot.final.vcf.gz",
14         "12": "/freebayes.chr_12.sites.snv_indel.annot.final.vcf.gz",
15         "13": "/freebayes.chr_13.sites.snv_indel.annot.final.vcf.gz",
16         "14": "/freebayes.chr_14.sites.snv_indel.annot.final.vcf.gz",
17         "15": "/freebayes.chr_15.sites.snv_indel.annot.final.vcf.gz",
18         "16": "/freebayes.chr_16.sites.snv_indel.annot.final.vcf.gz",
19         "17": "/freebayes.chr_17.sites.snv_indel.annot.final.vcf.gz",
20         "18": "/freebayes.chr_18.sites.snv_indel.annot.final.vcf.gz",
21         "19": "/freebayes.chr_19.sites.snv_indel.annot.final.vcf.gz",
22         "20": "/freebayes.chr_20.sites.snv_indel.annot.final.vcf.gz",
23         "21": "/freebayes.chr_21.sites.snv_indel.annot.final.vcf.gz",
24         "22": "/freebayes.chr_22.sites.snv_indel.annot.final.vcf.gz",
25         "X": "/freebayes.chr_X.sites.snv_indel.annot.final.vcf.gz",
26         "Y": "/freebayes.chr_Y.sites.snv_indel.annot.final.vcf.gz"
27     },
28     "results_base_path":
29     ↵   "/shared/data/results/regenotype_freebayes_discovery/",
30     "results_local_path": "/tmp/regenotype_freebayes_discovery/",
31     "freebayes": {
32         "mode": "regenotyping",
33         "flags": "-l"
34     }
35 }
```

Listing 29: Butler Workflow configuration for Data Submission.

```
1  {
2      "gnos": {
3          "ebi": {
4              "url": "https://gtrepo-ebi.annailabs.com"
5          },
6          "osdc_icgc": {
7              "url": "https://gtrepo-osdc-icgc.annailabs.com"
8          },
9          "osdc_tcga": {
10             "url": "https://gtrepo-osdc-tcga.annailabs.com"
11         }
12     },
13     "rsync": {
14         "flags": "-a -v --remove-source-files"
15     }
16 }
```

Listing 30: Butler Analysis configuration for Data Submission.

```

1  {
2      "gnos": {
3          "ebi": {
4              "key_location":
5                  ↳ "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.pem"
6          },
7          "osdc_icgc": {
8              "key_location":
9                  ↳ "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.pem"
10         },
11         "osdc_tcga": {
12             "key_location":
13                 ↳ "/home/airflow/.ssh/sergei_bionimbus_gnos_may.pem"
14         }
15     },
16     "metadata_template_location": "/opt/pcawg-germline/workflows/gtupload-wo_"
17     ↳ rkflow/analysis_template.xml",
18     "submission_base_path":
19     ↳ "/shared/data/results/freebayes_discovery_gnos_submission/",
20     "destination_repo_mapping": {
21         "ICGC": "ebi",
22         "TCGA": "osdc_tcga"
23     }
24 }
```

Listing 31: Python code for the run_delly function which implements the functionality of the delly_genotype task inside the Butler Delly Workflow.

```

1 def run_delly(**kwargs):
2
3     config = get_config(kwargs)
4     sample = get_sample(kwargs)
5
6     sample_id = sample["sample_id"]
7     sample_location = sample["sample_location"]
8
9     result_path_prefix = config["results_local_path"] + "/" + sample_id
10
11    if (not os.path.isdir(result_path_prefix)):
12        logger.info(
13            "Results directory {} not present,
14            ↳ creating.".format(result_path_prefix))
15        os.makedirs(result_path_prefix)
16
17    delly_path = config["delly"]["path"]
18    reference_location = config["reference_location"]
```

```
18     variants_location = config["variants_location"]
19     variants_type = config["variants_type"]
20     exclude_template_path = config["delly"]["exclude_template_path"]
21
22     result_filename = "{}_{}_{}.bcf".format(
23         result_path_prefix, sample_id, variants_type)
24
25     log_filename = "{}_{}_{}.log".format(
26         result_path_prefix, sample_id, variants_type)
27
28     delly_command = "{} call -t {} -g {} -v {} -o {} -x {} {} > {}".\
29         format(delly_path,
30                variants_type,
31                reference_location,
32                variants_location,
33                result_filename,
34                exclude_template_path,
35                sample_location,
36                log_filename)
37
38     call_command(delly_command, "delly")
39
40     copy_result(result_filename, sample_id, config)
```

Listing 32: Butler Delly Workflow analysis configuration to genotype deletions.

```
1 {
2     "variants_location": "/delly_deletion_sites/del.sites.bcf",
3     "results_base_path":
4         "/shared/data/results/delly_germline_deletions_14_07_2016/",
5     "results_local_path": "/tmp/delly_germline_deletions/",
6     "variants_type": "DEL"
7 }
```

Listing 33: Example of a VCF file (from <https://samtools.github.io/hts-specs/VCFv4.3.pdf>).

```
1 {
2     ##fileformat=VCFv4.3
3     ##fileDate=20090805
4     ##source=myImputationProgramV3.1
5     ##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
6     ##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb>
7         ↳ 2da,species="Homo
8         ↳ sapiens",taxonomy=x>
```

```

7  ##phasing=partial
8  ##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
9  ##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
10 ##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
11 ##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
12 ##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
13 ##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
14 ##FILTER=<ID=q10,Description="Quality below 10">
15 ##FILTER=<ID=s50,Description="Less than 50% of samples have data">
16 ##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
17 ##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
18 ##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
19 ##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
20 #CHROM POS ID REF ALT QUAL FILTER INFO FORMAT NA00001 NA00002 NA00003
21 20 14370 rs6054257 G A 29 PASS NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ
   ↳ 0|0:48:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
22 20 17330 . T A 3 q10 NS=3;DP=11;AF=0.017 GT:GQ:DP:HQ 0|0:49:3:58,50
   ↳ 0|1:3:5:65,3 0/0:41:3
23 20 1110696 rs6040355 A G,T 67 PASS NS=2;DP=10;AF=0.333,0.667;AA=T;DB
   ↳ GT:GQ:DP:HQ 1|2:21:6:23,27 2|1:2:0:18,2 2/2:35:4
24 20 1230237 . T . 47 PASS NS=3;DP=13;AA=T GT:GQ:DP:HQ 0|0:54:7:56,60
   ↳ 0|0:48:4:51,51 0/0:61:2
25 20 1234567 microsat1 GTC G,GTCT 50 PASS NS=3;DP=9;AA=G GT:GQ:DP 0/1:35:4
   ↳ 0/2:17:2 1/1:40:3

```

Bibliography

- [1] URL: <http://www.langmead-lab.org/teaching-materials/>.
- [2] URL: <https://software.broadinstitute.org/gatk/documentation>.
- [3] URL: <https://www.influxdata.com/time-series-platform/telegraf/>.
- [4] URL: <https://github.com/etsy/statsd>.
- [5] URL: <https://github.com/influxdata/kapacitor>.
- [6] URL: <https://slack.com>.
- [7] URL: <https://github.com/influxdata/chronograf>.
- [8] URL: <http://www.langmead-lab.org/teaching-materials/>.
- [9] .
- [10] Enis Afgan et al. “Galaxy CloudMan: delivering cloud compute clusters”. In: *BMC bioinformatics* 11.12 (2010), p. 1.
- [11] Charu C Aggarwal. *Data streams: models and algorithms*. Vol. 31. Springer Science & Business Media, 2007.
- [12] Daniel Aird et al. “Analyzing and minimizing PCR amplification bias in Illumina sequencing libraries”. In: *Genome biology* 12.2 (2011), R18.
- [13] Tyler S Alioto et al. “A comprehensive assessment of somatic mutation detection in cancer using whole-genome sequencing”. In: *Nature communications* 6 (2015), p. 10001.
- [14] Can Alkan, Bradley P Coe, and Evan E Eichler. “Genome structural variation discovery and genotyping”. In: *Nature Reviews Genetics* 12.5 (2011), p. 363.
- [15] Carmen J Allegra et al. “American Society of Clinical Oncology provisional clinical opinion: testing for KRAS gene mutations in patients with metastatic colorectal carcinoma to predict response to anti–epidermal growth factor receptor monoclonal antibody therapy”. In: *Journal of clinical oncology* 27.12 (2009), pp. 2091–2096.
- [16] Simon Andrews et al. “FastQC: a quality control tool for high throughput sequence data”. In: (2010).
- [17] *Apache Airflow (incubating) Documentation — Airflow Documentation*. URL: <https://airflow.incubator.apache.org/> (visited on 10/31/2016).
- [18] Brian Babcock et al. “Models and issues in data stream systems”. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2002, pp. 1–16.

- [19] Shay Banon. *Elasticsearch*. URL: <https://www.elastic.co/products/elasticsearch> (visited on 10/13/2016).
- [20] Ralph M Barnes. “Motion and time study.” In: (1949).
- [21] James O Berger. *Statistical decision theory and Bayesian analysis*. Springer Science & Business Media, 2013.
- [22] David Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.
- [23] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. “Trimmomatic: a flexible trimmer for Illumina sequence data”. In: *Bioinformatics* 30.15 (2014), pp. 2114–2120.
- [24] Léon Bottou. “Large-scale machine learning with stochastic gradient descent”. In: *Proceedings of COMPSTAT’2010*. Springer, 2010, pp. 177–186.
- [25] Kym M Boycott et al. “Rare-disease genetics in the era of next-generation sequencing: discovery to translation”. In: *Nature Reviews Genetics* 14.10 (2013), pp. 681–691.
- [26] Benoit G Bruneau. “The developmental genetics of congenital heart disease”. In: *Nature* 451.7181 (2008), pp. 943–948.
- [27] Michael Burrows and David J Wheeler. “A block-sorting lossless data compression algorithm”. In: (1994).
- [28] Rajkumar Buyya et al. “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”. In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.
- [29] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.
- [30] Xiaoyu Chen et al. “Manta: rapid detection of structural variants and indels for germline and cancer sequencing applications”. In: *Bioinformatics* 32.8 (2015), pp. 1220–1222.
- [31] Kristian Cibulskis et al. “ContEst: estimating cross-contamination of human samples in next-generation sequencing data”. In: *Bioinformatics* 27.18 (2011), pp. 2601–2602.
- [32] Kristian Cibulskis et al. “Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples”. In: *Nature biotechnology* 31.3 (2013), pp. 213–219.
- [33] John G Cleary et al. “Joint variant and de novo mutation identification on pedigrees from high-throughput sequencing data”. In: *Journal of Computational Biology* 21.6 (2014), pp. 405–419.
- [34] Peter JA Cock et al. “The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants”. In: *Nucleic acids research* 38.6 (2009), pp. 1767–1771.
- [35] Francis S Collins and Harold Varmus. “A new initiative on precision medicine”. In: *New England Journal of Medicine* 372.9 (2015), pp. 793–795.
- [36] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. “How to apply de Bruijn graphs to genome assembly”. In: *Nature biotechnology* 29.11 (2011), p. 987.

- [37] 1000 Genomes Project Consortium et al. “A map of human genome variation from population-scale sequencing”. In: *Nature* 467.7319 (2010), pp. 1061–1073.
- [38] 1000 Genomes Project Consortium et al. “An integrated map of genetic variation from 1,092 human genomes”. In: *Nature* 491.7422 (2012), pp. 56–65.
- [39] ENCODE Project Consortium et al. “An integrated encyclopedia of DNA elements in the human genome”. In: *Nature* 489.7414 (2012), pp. 57–74.
- [40] International HapMap Consortium et al. “The international HapMap project”. In: *Nature* 426.6968 (2003), p. 789.
- [41] Wellcome Trust Case Control Consortium et al. “Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls”. In: *Nature* 447.7145 (2007), p. 661.
- [42] Charles E Cook et al. “The european bioinformatics institute in 2016: Data growth and integration”. In: *Nucleic acids research* 44.D1 (2016), pp. D20–D26.
- [43] Georgiana Copil et al. “Multi-level elasticity control of cloud services”. In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 429–436.
- [44] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.
- [45] Antonio Cuevas, Manuel Febrero, and Ricardo Fraiman. “Cluster analysis: a further approach based on density estimation”. In: *Computational Statistics & Data Analysis* 36.4 (2001), pp. 441–459.
- [46] Vasa Curcin and Moustafa Ghanem. “Scientific workflow systems-can one size fit all?” In: *2008 Cairo International Biomedical Engineering Conference*. IEEE. 2008, pp. 1–9.
- [47] Armon Dadgar. *Consul*. HashiCorp. URL: <https://www.consul.io/> (visited on 10/13/2016).
- [48] Mark J Daly et al. “High-resolution haplotype structure in the human genome”. In: *Nature genetics* 29.2 (2001), p. 229.
- [49] Petr Danecek et al. “The variant call format and VCFtools”. In: *Bioinformatics* 27.15 (2011), pp. 2156–2158.
- [50] Charles Darwin. *On the origin of species*. D. Appleton and Co., 1871. DOI: 10.5962/bhl.title.28875.
- [51] Mayur Datar et al. “Maintaining stream statistics over sliding windows”. In: *SIAM journal on computing* 31.6 (2002), pp. 1794–1813.
- [52] J Davis II et al. *Overview of the Ptolemy project*. Tech. rep. ERL Technical Report UCB/ERL, 1999.
- [53] Olivier Delaneau, Jonathan Marchini, and Jean-François Zagury. “A linear complexity phasing method for thousands of genomes”. In: *Nature methods* 9.2 (2012), pp. 179–181.
- [54] Mark A DePristo et al. “A framework for variation discovery and genotyping using next-generation DNA sequencing data”. In: *Nature genetics* 43.5 (2011), pp. 491–498.

- [55] Wil MP van Der Aalst et al. “Workflow patterns”. In: *Distributed and parallel databases* 14.1 (2003), pp. 5–51.
- [56] Julianne C Dohm et al. “Substantial biases in ultra-short read data sets from high-throughput DNA sequencing”. In: *Nucleic acids research* 36.16 (2008), e105–e105.
- [57] Richard Durbin et al. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.
- [58] Mark TW Ebbert et al. “Evaluating the necessity of PCR duplicate removal from next-generation sequencing data and a comparison of approaches”. In: *BMC bioinformatics* 17.7 (2016), p. 239.
- [59] Genomics England. “The 100,000 genomes project”. In: *The 100* (2016), pp. 1–2.
- [60] Thomas Erl. *Service-oriented architecture (SOA): concepts, technology, and design*. 2005.
- [61] Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action*. Manning Greenwich, 2011.
- [62] Patrick Th Eugster et al. “The many faces of publish/subscribe”. In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.
- [63] European Open Science Cloud / Open Science - Research and Innovation - European Commission. URL: <http://ec.europa.eu/research/open-science/index.cfm?pg=open-science-cloud> (visited on 10/31/2016).
- [64] Kelly R Ewen et al. “Identification and analysis of error types in high-throughput genotyping”. In: *The American Journal of Human Genetics* 67.3 (2000), pp. 727–736.
- [65] Warren J Ewens. “The sampling theory of selectively neutral alleles”. In: *Theoretical population biology* 3.1 (1972), pp. 87–112.
- [66] Michael Factor et al. “Object storage: The future building block for storage systems”. In: *2005 IEEE International Symposium on Mass Storage Systems and Technology*. IEEE. 2005, pp. 119–123.
- [67] Michael Farrar. “Striped Smith–Waterman speeds database searches six times over other SIMD implementations”. In: *Bioinformatics* 23.2 (2006), pp. 156–161.
- [68] Gregory G Faust and Ira M Hall. “YAHA: fast and flexible long-read alignment with optimal breakpoint detection”. In: *Bioinformatics* 28.19 (2012), pp. 2417–2424.
- [69] Paolo Ferragina and Giovanni Manzini. “Opportunistic data structures with applications”. In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE. 2000, pp. 390–398.
- [70] Roy Thomas Fielding. “Architectural styles and the design of network-based software architectures”. PhD thesis. University of California, Irvine, 2000.
- [71] Simon A Forbes et al. “COSMIC: exploring the world’s knowledge of somatic mutations in human cancer”. In: *Nucleic acids research* 43.D1 (2015), pp. D805–D811.

- [72] G David Forney. “The viterbi algorithm”. In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278.
- [73] Rosalind E Franklin and Raymond G Gosling. “Molecular configuration in sodium thymonucleate”. In: *Nature* 171 (1953), pp. 740–741.
- [74] Markus Hsi-Yang Fritz et al. “Efficient storage of high throughput DNA sequencing data using reference-based compression”. In: *Genome research* 21.5 (2011), pp. 734–740.
- [75] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. “Mining data streams: a review”. In: *ACM Sigmod Record* 34.2 (2005), pp. 18–26.
- [76] Hector Garcia-Molina, Frank Germano, and Walter H Kohler. “Debugging a distributed computing system”. In: *IEEE Transactions on Software Engineering* 2 (1984), pp. 210–219.
- [77] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016.
- [78] Erik Garrison and Gabor Marth. “Haplotype-based variant detection from short-read sequencing”. In: *arXiv preprint arXiv:1207.3907* (2012).
- [79] *Glossary V1 - EGIWiki*. URL: https://wiki.egi.eu/wiki/Glossary_V1 (visited on 10/28/2016).
- [80] Jeremy Goecks, Anton Nekrutenko, and James Taylor. “Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences”. In: *Genome biology* 11.8 (2010), p. 1.
- [81] A Gordon and GJ Hannon. “Fastx-toolkit”. In: *FASTQ/A short-reads preprocessing tools (unpublished)* http://hannonlab.cshl.edu/fastx_toolkit 5 (2010).
- [82] Christopher Greenman et al. “Patterns of somatic mutation in human cancer genomes”. In: *Nature* 446.7132 (2007), pp. 153–158.
- [83] Michael Greenwald and Sanjeev Khanna. “Space-efficient online computation of quantile summaries”. In: *ACM SIGMOD Record*. Vol. 30. 2. ACM. 2001, pp. 58–66.
- [84] Mendel Gregor. “Versuche über Pflanzen-Hybriden. Verhandlungen des naturforschenden Vereines in Brunn”. In: 4 (1865), pp. 3–47.
- [85] Robert L Grossman et al. “An overview of the open science data cloud”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM. 2010, pp. 377–384.
- [86] Daniel F Gudbjartsson et al. “Large-scale whole-genome sequencing of the Icelandic population”. In: *Nature genetics* 47.5 (2015), pp. 435–444.
- [87] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.
- [88] Douglas Hanahan and Robert A Weinberg. “Hallmarks of cancer: the next generation”. In: *cell* 144.5 (2011), pp. 646–674.
- [89] John A Hartigan. “Clustering algorithms”. In: (1975).

- [90] Michael J Heller. “DNA microarray technology: devices, systems, and applications”. In: *Annual review of biomedical engineering* 4.1 (2002), pp. 129–153.
- [91] Robert L Henderson. “Job scheduling under the portable batch system”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 279–294.
- [92] Joel N Hirschhorn and Mark J Daly. “Genome-wide association studies for common diseases and complex traits”. In: *Nature Reviews Genetics* 6.2 (2005), p. 95.
- [93] Alan Hodgkinson and Adam Eyre-Walker. “Human triallelic sites: evidence for a new mutational mechanism?” In: *Genetics* 184.1 (2010), pp. 233–241.
- [94] David Hollingsworth. “The workflow reference model”. In: (1995).
- [95] *Home*. URL: <https://www.denbi.de/> (visited on 10/31/2016).
- [96] *Homepage / Celery: Distributed Task Queue*. URL: <http://www.celeryproject.org/> (visited on 10/31/2016).
- [97] Eun Pyo Hong and Ji Wan Park. “Sample size and statistical power calculation in genetic association studies”. In: *Genomics & informatics* 10.2 (2012), pp. 117–122.
- [98] *InfluxDB – Time-Series Data Storage*. InfluxData. URL: <https://www.influxdata.com/time-series-platform/influxdb/> (visited on 10/12/2016).
- [99] David B Ingham, Fabio Panzieri, and Santosh K Shrivastava. “Constructing dependable Web services”. In: *Advances in Distributed Systems*. Springer, 2000, pp. 277–294.
- [100] Vernon M Ingram et al. “Gene mutations in human haemoglobin: the chemical difference between normal and sickle cell haemoglobin”. In: *Nature* 180.4581 (1957), pp. 326–328.
- [101] Zamin Iqbal et al. “De novo assembly and genotyping of variants using colored de Bruijn graphs”. In: *Nature genetics* 44.2 (2012), p. 226.
- [102] Mark Jobling, Matthew Hurles, and Chris Tyler-Smith. *Human evolutionary genetics: origins, peoples & disease*. Garland Science, 2013.
- [103] Peter A Jones and Stephen B Baylin. “The epigenomics of cancer”. In: *Cell* 128.4 (2007), pp. 683–692.
- [104] Jocelyn Kaiser and Jennifer Couzin-Frankel. “Biden seeks clear course for his cancer moonshot”. In: *Science* 351.6271 (2016), pp. 325–326.
- [105] Juha Kärkkäinen and Peter Sanders. “Simple linear work suffix array construction”. In: *International Colloquium on Automata, Languages, and Programming*. Springer, 2003, pp. 943–955.
- [106] *Kibana*. URL: <https://www.elastic.co/products/kibana> (visited on 10/13/2016).
- [107] Daehwan Kim et al. “Centrifuge: rapid and sensitive classification of metagenomic sequences”. In: *Genome research* (2016).

- [108] Bartha Maria Knoppers and Ruth Chadwick. “Human genetic research: emerging trends in ethics”. In: *Nature Reviews Genetics* 6.1 (2005), pp. 75–79.
- [109] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. 2011, pp. 1–7.
- [110] Hans-Peter Kriegel et al. “Density-based clustering”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1.3 (2011), pp. 231–240.
- [111] Eric S Lander et al. “Initial sequencing and analysis of the human genome”. In: *Nature* 409.6822 (2001), pp. 860–921.
- [112] Ben Langmead and Steven L Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature methods* 9.4 (2012), pp. 357–359.
- [113] Ben Langmead et al. “Searching for SNPs with cloud computing”. In: *Genome biology* 10.11 (2009), R134.
- [114] Ben Langmead et al. “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”. In: *Genome biology* 10.3 (2009), R25.
- [115] Martin Lauss et al. “Monitoring of technical variation in quantitative high-throughput datasets”. In: *Cancer informatics* 12 (2013), CIN-S12862.
- [116] Michael S Lawrence et al. “Discovery and saturation analysis of cancer genes across 21 tumour types”. In: *Nature* 505.7484 (2014), pp. 495–501.
- [117] Ryan M Layer et al. “LUMPY: a probabilistic framework for structural variant discovery”. In: *Genome biology* 15.6 (2014), R84.
- [118] Hane Lee et al. “Clinical exome sequencing for genetic identification of rare Mendelian disorders”. In: *Jama* 312.18 (2014), pp. 1880–1887.
- [119] Monkol Lek et al. “Analysis of protein-coding genetic variation in 60,706 humans”. In: *Nature* 536.7616 (2016), pp. 285–291.
- [120] Reuven M Lerner. “At the forge: PostgreSQL, the NoSQL database”. In: *Linux Journal* 2014.247 (2014), p. 5.
- [121] Heng Li. “A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data”. In: *Bioinformatics* 27.21 (2011), pp. 2987–2993.
- [122] Heng Li. “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”. In: *arXiv preprint arXiv:1303.3997* (2013).
- [123] Heng Li. “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”. In: *arXiv preprint arXiv:1303.3997* (2013).
- [124] Heng Li. “Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly”. In: *Bioinformatics* 28.14 (2012), pp. 1838–1844.
- [125] Heng Li. “Mathematical Notes on SAMtools Algorithms”. In: (2010).
- [126] Heng Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 1 (2018), p. 7.
- [127] Heng Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 1 (2018), p. 7.

- [128] Heng Li. “Tabix: fast retrieval of sequence features from generic TAB-delimited files”. In: *Bioinformatics* 27.5 (2011), pp. 718–719.
- [129] Heng Li. “Towards better understanding of artifacts in variant calling from high-coverage samples”. In: *Bioinformatics* (2014), btu356.
- [130] Heng Li and Richard Durbin. “Fast and accurate long-read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 26.5 (2010), pp. 589–595.
- [131] Heng Li and Richard Durbin. “Fast and accurate short read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760.
- [132] Heng Li and Nils Homer. “A survey of sequence alignment algorithms for next-generation sequencing”. In: *Briefings in bioinformatics* 11.5 (2010), pp. 473–483.
- [133] Heng Li, Jue Ruan, and Richard Durbin. “Mapping short DNA sequencing reads and calling variants using mapping quality scores”. In: *Genome research* (2008), gr-078212.
- [134] Heng Li et al. “A synthetic-diploid benchmark for accurate variant-calling evaluation”. In: *Nature methods* 15.8 (2018), p. 595.
- [135] Heng Li et al. “The sequence alignment/map format and SAMtools”. In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.
- [136] Ruiqiang Li et al. “SOAP: short oligonucleotide alignment program”. In: *Bioinformatics* 24.5 (2008), pp. 713–714.
- [137] Hengyun Lu, Francesca Giordano, and Zemin Ning. “Oxford Nanopore MinION sequencing and genome assembly”. In: *Genomics, proteomics & bioinformatics* 14.5 (2016), pp. 265–279.
- [138] Bertram Ludäscher et al. “Scientific workflow management and the Kepler system”. In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.
- [139] Ramon Luengo-Fernandez et al. “Economic burden of cancer across the European Union: a population-based cost analysis”. In: *The lancet oncology* 14.12 (2013), pp. 1165–1174.
- [140] David Malkin et al. “Germ line p53 mutations in a familial syndrome of breast cancer, sarcomas, and other neoplasms”. In: *Science* (1990), pp. 1233–1238.
- [141] Udi Manber and Gene Myers. “Suffix arrays: a new method for on-line string searches”. In: *siam Journal on Computing* 22.5 (1993), pp. 935–948.
- [142] Teri A Manolio. “Genomewide association studies and assessment of the risk of disease”. In: *New England Journal of Medicine* 363.2 (2010), pp. 166–176.
- [143] Ming Mao and Marty Humphrey. “Auto-scaling to minimize cost and meet application deadlines in cloud workflows”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE. 2011, pp. 1–12.
- [144] Elaine R Mardis. “Next-generation DNA sequencing methods”. In: *Annu. Rev. Genomics Hum. Genet.* 9 (2008), pp. 387–402.

- [145] Ronald Margolis et al. “The National Institutes of Health’s Big Data to Knowledge (BD2K) initiative: capitalizing on biomedical big data”. In: *Journal of the American Medical Informatics Association* 21.6 (2014), pp. 957–958.
- [146] Gabor T Marth et al. “A general approach to single-nucleotide polymorphism discovery”. In: *Nature genetics* 23.4 (1999), p. 452.
- [147] Vivien Marx. “Biology: The big challenges of big data”. In: *Nature* 498.7453 (2013), pp. 255–260.
- [148] Vivien Marx. “The DNA of a nation”. In: *Nature* 524.7566 (2015), pp. 503–505.
- [149] Aaron McKenna et al. “The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data”. In: *Genome research* 20.9 (2010), pp. 1297–1303.
- [150] William McLaren et al. “The ensembl variant effect predictor”. In: *Genome biology* 17.1 (2016), p. 122.
- [151] Peter Mell and Tim Grance. “The NIST definition of cloud computing”. In: (2011).
- [152] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.
- [153] C Mohinudeen et al. “An Overview of Next-Generation Sequencing (NGS) Technologies to Study the Molecular Diversity of Genome”. In: *Microbial Applications Vol. 1*. Springer, 2017, pp. 295–317.
- [154] Fruzsina Molnár-Gábor et al. “Computing patient data in the cloud: practical and legal considerations for genetics and genomics research in Europe and internationally”. In: *Genome Medicine* 9.1 (2017), p. 58.
- [155] Todd K Moon. “The expectation-maximization algorithm”. In: *IEEE Signal processing magazine* 13.6 (1996), pp. 47–60.
- [156] Paul Muir et al. “The real cost of sequencing: scaling computation to keep pace with data generation”. In: *Genome biology* 17.1 (2016), p. 53.
- [157] Shanmugavelayutham Muthukrishnan et al. “Data streams: Algorithms and applications”. In: *Foundations and Trends® in Theoretical Computer Science* 1.2 (2005), pp. 117–236.
- [158] Eugene W Myers and Webb Miller. “Optimal alignments in linear space”. In: *Bioinformatics* 4.1 (1988), pp. 11–17.
- [159] Michael W Nachman. “Single nucleotide polymorphisms and recombination rate in humans”. In: *TRENDS in Genetics* 17.9 (2001), pp. 481–485.
- [160] Rasmus Nielsen et al. “Genotype and SNP calling from next-generation sequencing data”. In: *Nature Reviews Genetics* 12.6 (2011), pp. 443–451.
- [161] Bill Nitzberg and Virginia Lo. “Distributed shared memory: A survey of issues and algorithms”. In: *Distributed Shared Memory-Concepts and Systems* (1991), pp. 42–50.
- [162] Patrick O’Neil et al. “The log-structured merge-tree (LSM-tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385.

- [163] Torkel Ödegaard. *Grafana.net*. URL: <https://grafana.net/> (visited on 10/12/2016).
- [164] Tom Oinn et al. “Taverna: a tool for the composition and enactment of bioinformatics workflows”. In: *Bioinformatics* 20.17 (2004), pp. 3045–3054.
- [165] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 305–319.
- [166] David Oppenheimer and David A Patterson. “Architecture and dependability of large-scale internet services”. In: *IEEE Internet Computing* 6.5 (2002), pp. 41–49.
- [167] Stavros Papadopoulos et al. “The TileDB array data storage manager”. In: *Proceedings of the VLDB Endowment* 10.4 (2016), pp. 349–360.
- [168] Mike P Papazoglou. “Service-oriented computing: Concepts, characteristics and directions”. In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE. 2003, pp. 3–12.
- [169] Emanuel Parzen. “On estimation of a probability density function and mode”. In: *The annals of mathematical statistics* 33.3 (1962), pp. 1065–1076.
- [170] Ravi K Patel and Mukesh Jain. “NGS QC Toolkit: a toolkit for quality control of next generation sequencing data”. In: *PloS one* 7.2 (2012), e30619.
- [171] Jaume Pellicer, Michael F Fay, and Ilia J Leitch. “The largest eukaryotic genome of them all?” In: *Botanical Journal of the Linnean Society* 164.1 (2010), pp. 10–15.
- [172] James L Peterson. “Petri net theory and the modeling of systems”. In: (1981).
- [173] *Picard toolkit*. <http://broadinstitute.github.io/picard/>. 2018.
- [174] Erin D Pleasance et al. “A comprehensive catalogue of somatic mutations from a human cancer genome”. In: *Nature* 463.7278 (2010), p. 191.
- [175] Aaron R Quinlan and Ira M Hall. “BEDTools: a flexible suite of utilities for comparing genomic features”. In: *Bioinformatics* 26.6 (2010), pp. 841–842.
- [176] *RabbitMQ - Messaging that just works*. URL: <https://www.rabbitmq.com/> (visited on 10/31/2016).
- [177] Tobias Rausch et al. “DELLY: structural variant discovery by integrated paired-end and split-read analysis”. In: *Bioinformatics* 28.18 (2012), pp. i333–i339.
- [178] Knut Reinert et al. “Alignment of next-generation sequencing reads”. In: *Annual review of genomics and human genetics* 16 (2015), pp. 133–151.
- [179] Anthony Rhoads and Kin Fai Au. “PacBio sequencing and its applications”. In: *Genomics, proteomics & bioinformatics* 13.5 (2015), pp. 278–289.
- [180] Andy Rimmer et al. “Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications”. In: *Nature genetics* 46.8 (2014), pp. 912–918.
- [181] Michael Roberts et al. “Reducing storage requirements for biological sequence comparison”. In: *Bioinformatics* 20.18 (2004), pp. 3363–3369.

- [182] Nicola D Roberts et al. “A comparative analysis of algorithms for somatic SNV detection in cancer”. In: *Bioinformatics* (2013), btt375.
- [183] Richard J Roberts, Mauricio O Carneiro, and Michael C Schatz. “The advantages of SMRT sequencing”. In: *Genome biology* 14.6 (2013), p. 405.
- [184] Dan Robinson et al. “Integrative clinical genomics of advanced prostate cancer”. In: *Cell* 161.5 (2015), pp. 1215–1228.
- [185] Murray Rosenblatt. “Remarks on some nonparametric estimates of a density function”. In: *The Annals of Mathematical Statistics* (1956), pp. 832–837.
- [186] Richard M Russell. “The CRAY-1 computer system”. In: *Communications of the ACM* 21.1 (1978), pp. 63–72.
- [187] *SaltStack automation for CloudOps, ITOps, and DevOps at scale*. URL: <https://saltstack.com/> (visited on 10/31/2016).
- [188] Fred Sanger and Alan R Coulson. “A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase”. In: *Journal of molecular biology* 94.3 (1975), 441IN19447–446IN20448.
- [189] Frederick Sanger, Steven Nicklen, and Alan R Coulson. “DNA sequencing with chain-terminating inhibitors”. In: *Proceedings of the national academy of sciences* 74.12 (1977), pp. 5463–5467.
- [190] Wataru Satake et al. “Genome-wide association study identifies common variants at four loci as genetic risk factors for Parkinson’s disease”. In: *Nature genetics* 41.12 (2009), p. 1303.
- [191] Nitin Sawant and Himanshu Shah. “Big Data Access Patterns”. In: *Big Data Application Architecture Q & A*. Springer, 2013, pp. 57–68.
- [192] Stephan C Schuster. “Next-generation sequencing transforms today’s biology”. In: *Nature* 200.8 (2007), pp. 16–18.
- [193] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. “OpenStack: toward an open-source solution for cloud computing”. In: *International Journal of Computer Applications* 55.3 (2012).
- [194] Dennis J Selkoe. “Amyloid β -protein and the genetics of Alzheimer’s disease”. In: *Journal of Biological Chemistry* 271.31 (1996), pp. 18295–18298.
- [195] Robert Shapiro. “A technical comparison of XPDL, BPML and BPEL4WS”. In: *Cape Visions* (2002).
- [196] Mary Shaw and David Garlan. *Software architecture*. Vol. 101. Prentice Hall Englewood Cliffs, 1996.
- [197] Hui Shen et al. “Comprehensive characterization of human genome variation by high coverage whole-genome sequencing of forty four Caucasians”. In: *PLoS One* 8.4 (2013), e59494.
- [198] Nisheeth Shrivastava et al. “Medians and beyond: new aggregation techniques for sensor networks”. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 239–249.
- [199] Jared T Simpson and Richard Durbin. “Efficient construction of an assembly string graph using the FM-index”. In: *Bioinformatics* 26.12 (2010), pp. i367–i373.

- [200] Jared T Simpson and Richard Durbin. “Efficient de novo assembly of large genomes using compressed data structures”. In: *Genome research* 22.3 (2012), pp. 549–556.
- [201] Jordan Sissel. *Logstash*. URL: <https://www.elastic.co/products/logstash> (visited on 10/13/2016).
- [202] Temple F Smith and Michael S Waterman. “Comparison of biosequences”. In: *Advances in applied mathematics* 2.4 (1981), pp. 482–489.
- [203] *Start page – collectd – The system statistics collection daemon*. URL: <https://collectd.org/> (visited on 10/31/2016).
- [204] Lincoln D Stein et al. “Data analysis: create a cloud commons”. In: *Nature* 523 (2015), pp. 149–151.
- [205] Zachary D Stephens et al. “Big data: astronomical or genomic”. In: *PLoS biology* 13.7 (2015), e1002195.
- [206] Zachary D Stephens et al. “Simulating next-generation sequencing datasets from empirical mutation and sequencing models”. In: *PloS one* 11.11 (2016), e0167047.
- [207] Michael R Stratton, Peter J Campbell, and P Andrew Futreal. “The cancer genome”. In: *Nature* 458.7239 (2009), pp. 719–724.
- [208] Peter H Sudmant et al. “An integrated map of structural variation in 2,504 human genomes”. In: *Nature* 526.7571 (2015), pp. 75–81.
- [209] Adrian Tan, Gonçalo R Abecasis, and Hyun Min Kang. “Unified representation of genetic variants”. In: *Bioinformatics* (2015), btv112.
- [210] Frederick Winslow Taylor. *Scientific management*. Routledge, 2004.
- [211] *Terraform by HashiCorp*. URL: <https://www.terraform.io/> (visited on 10/31/2016).
- [212] *The Cost of Sequencing a Human Genome - National Human Genome Research Institute (NHGRI)*. URL: <https://www.genome.gov/sequencingcosts/> (visited on 11/04/2016).
- [213] Lindsey A Torre et al. “Global cancer statistics, 2012”. In: *CA: a cancer journal for clinicians* 65.2 (2015), pp. 87–108.
- [214] Frances Susan Turner. “Assessment of insert sizes and adapter content in fastq data from NexteraXT libraries”. In: *Frontiers in genetics* 5 (2014), p. 5.
- [215] Esko Ukkonen. “On-line construction of suffix trees”. In: *Algorithmica* 14.3 (1995), pp. 249–260.
- [216] Geraldine A Van der Auwera et al. “From FastQ data to high-confidence variant calls: the genome analysis toolkit best practices pipeline”. In: *Current protocols in bioinformatics* (2013), pp. 11–10.
- [217] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. “Dynamically scaling applications in the cloud”. In: *ACM SIGCOMM Computer Communication Review* 41.1 (2011), pp. 45–52.
- [218] J Craig Venter et al. “The sequence of the human genome”. In: *science* 291.5507 (2001), pp. 1304–1351.

- [219] J Craig Venter et al. “Shotgun sequencing of the human genome”. In: *Science* 280.5369 (1998), pp. 1540–1542.
- [220] Kathleen A Vermeersch and Mark P Styczynski. “Applications of metabolomics in cancer research”. In: *Journal of carcinogenesis* 12 (2013).
- [221] Karl V Voelkerding, Shale A Dames, and Jacob D Durtschi. “Next-generation sequencing: from basic research to diagnostics”. In: *Clinical chemistry* 55.4 (2009), pp. 641–658.
- [222] Jeremiah A Wala et al. “SvABA: genome-wide detection of structural variants and indels by local assembly”. In: *Genome research* (2018).
- [223] David G Wang et al. “Large-scale identification, mapping, and genotyping of single-nucleotide polymorphisms in the human genome”. In: *Science* 280.5366 (1998), pp. 1077–1082.
- [224] Zhong Wang, Mark Gerstein, and Michael Snyder. “RNA-Seq: a revolutionary tool for transcriptomics”. In: *Nature reviews genetics* 10.1 (2009), pp. 57–63.
- [225] James D Watson, Francis HC Crick, et al. “Molecular structure of nucleic acids”. In: *Nature* 171.4356 (1953), pp. 737–738.
- [226] Sage A Weil et al. “Ceph: A scalable, high-performance distributed file system”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.
- [227] John N Weinstein et al. “The cancer genome atlas pan-cancer analysis project”. In: *Nature genetics* 45.10 (2013), pp. 1113–1120.
- [228] Jeffrey N Weitzel et al. “Genetics, genomics, and cancer risk assessment: state of the art and future directions in the era of personalized medicine”. In: *CA: a cancer journal for clinicians* 61.5 (2011), pp. 327–359.
- [229] BP Welford. “Note on a method for calculating corrected sums of squares and products”. In: *Technometrics* 4.3 (1962), pp. 419–420.
- [230] Danielle Welter et al. “The NHGRI GWAS Catalog, a curated resource of SNP-trait associations”. In: *Nucleic acids research* 42.D1 (2013), pp. D1001–D1006.
- [231] Justin P Whalley et al. “Framework For Quality Assessment Of Whole Genome, Cancer Sequences”. In: *bioRxiv* (2017), p. 140921.
- [232] Philipp Wieder et al. *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.
- [233] Christopher Wilks et al. “The Cancer Genomics Hub (CGHub): overcoming cancer through the power of torrential data”. In: *Database* 2014 (2014), bau093.
- [234] Derrick E Wood and Steven L Salzberg. “Kraken: ultrafast metagenomic sequence classification using exact alignments”. In: *Genome biology* 15.3 (2014), R46.
- [235] K Robin Yabroff et al. “Economic burden of cancer in the United States: estimates, projections, and future research”. In: *Cancer Epidemiology Biomarkers & Prevention* 20.10 (2011), pp. 2006–2014.

- [236] Yaping Yang et al. “Clinical whole-exome sequencing for the diagnosis of mendelian disorders”. In: *New England Journal of Medicine* 369.16 (2013), pp. 1502–1511.
- [237] Kai Ye et al. “Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads”. In: *Bioinformatics* 25.21 (2009), pp. 2865–2871.
- [238] Christina K Yung et al. “ICGC in the cloud”. In: *Cancer Research* 76.14 Supplement (2016), pp. 3605–3605.
- [239] Shelemyahu Zacks. *The theory of statistical inference*. Vol. 34. Wiley New York, 1971.
- [240] Matei Zaharia et al. “Faster and more accurate sequence alignment with SNAP”. In: *arXiv preprint arXiv:1111.5572* (2011).
- [241] Daniel R Zerbino and Ewan Birney. “Velvet: algorithms for de novo short read assembly using de Bruijn graphs”. In: *Genome research* 18.5 (2008), pp. 821–829.
- [242] Min Zhao et al. “Computational tools for copy number variation (CNV) detection using next-generation sequencing data: features and perspectives”. In: *BMC bioinformatics* 14.11 (2013), S1.
- [243] Qian Zhou et al. “QC-Chain: fast and holistic quality control method for next-generation sequencing data”. In: *PloS one* 8.4 (2013), e60234.
- [244] Songnian Zhou. “Lsf: Load sharing in large heterogeneous distributed systems”. In: *I Workshop on Cluster Computing*. Vol. 136. 1992.
- [245] Justin Zook et al. “Reproducible integration of multiple sequencing datasets to form high-confidence SNP, indel, and reference calls for five human genome reference materials”. In: *BioRxiv* (2018), p. 281006.