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

Sergei Yakneen
EMBL
Meyerhofstrasowse 1,
69117 Heidelberg, Germany

19/07/2016

Contents

Chapter 1

Introduction

1.1 Context and Motivation

In the 17 years since the publication of the first draft human genome[72] 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[123], and the development of the Internet and cloud computing[19]. 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[58], 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[67], to identify causes and mechanisms of action of rare genetic diseases that affect a very small proportion of the population[17], and to elucidate and potentially target the genetic component of more common diseases such as cancer[147], heart disease[18], or dementia[125] 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[76]. However, the other 99% of the human genome, it's non-coding regions, contain crucial information such as gene regulatory elements[26] 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 [129]. WGS studies at 100,000 participants scale that are planned for the coming years[40] 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[75] and certain types of cancers[117] where a small set of genomic loci is interrogated via a gene panel[10] 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[112], 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[130].

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[98].

Data Formats - The data formats used for storing genomic data are primarily large size character and binary files (FASTA, SAM, BAM, VCF)[82, 31] 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[96].

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[85] and somatic[55] genomic variants, transcriptomics[144], epigenomics[68], metabolomics[141], 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[117].

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[36]. 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[148]. 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[65], 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[142]. 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[80, 155, 9, 92] 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[95]. 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 ac-

curacy (by looking at less data, or using faster but less accurate algorithms[81]), by increasing the level of parallelisation within the computational pipeline i.e. parallelising steps that are currently sequential[73], 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[107] (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[91, 138] 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[41] 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[108]. 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[28]. 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[51]. 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[82, 49], and VCF[31], 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[39].

Rheos adopts a data and event stream approach to accomplish scalable fine-grained communication between services[99]. 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[59], 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[7]. 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[42]. 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[50]. Because genomic algorithms frequently make use of various summary statistics accumulated over the data-set[109, 78], 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[33, 12] 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[82, 52, 21, 115]. 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)[82]. 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[149], have been aligned[81], have been investigated for PCR duplicates[138], and have been sorted[138]. 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[153, 15].

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[82, 52]. 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[8]. 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[44], or contamination[20] 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[40, 69, 88] 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 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[32] and Gregor Mendel[56]. 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[48], James Watson, and Francis Crick[145] who first elucidated the physical structure of DNA, then followed by the work of Fred Sanger[121, 120] 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[110] 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[121]. 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*[154]. Once one such full sequence (known as a *reference* sequnce) 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*[80] 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)[100] 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[86].

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[72] - 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[139], using a new sequencing method called shotgun sequencing[140], a method that formed the basis for a new revolution

in sequencing technology, now termed Next Generation Sequencing[123].

2.1.2 Next Generation Sequencing

The Next Generation Sequencing methodology[87] 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[81]. Because NGS sequencing methods are prone to certain errors and biases[38], 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*[101], 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.

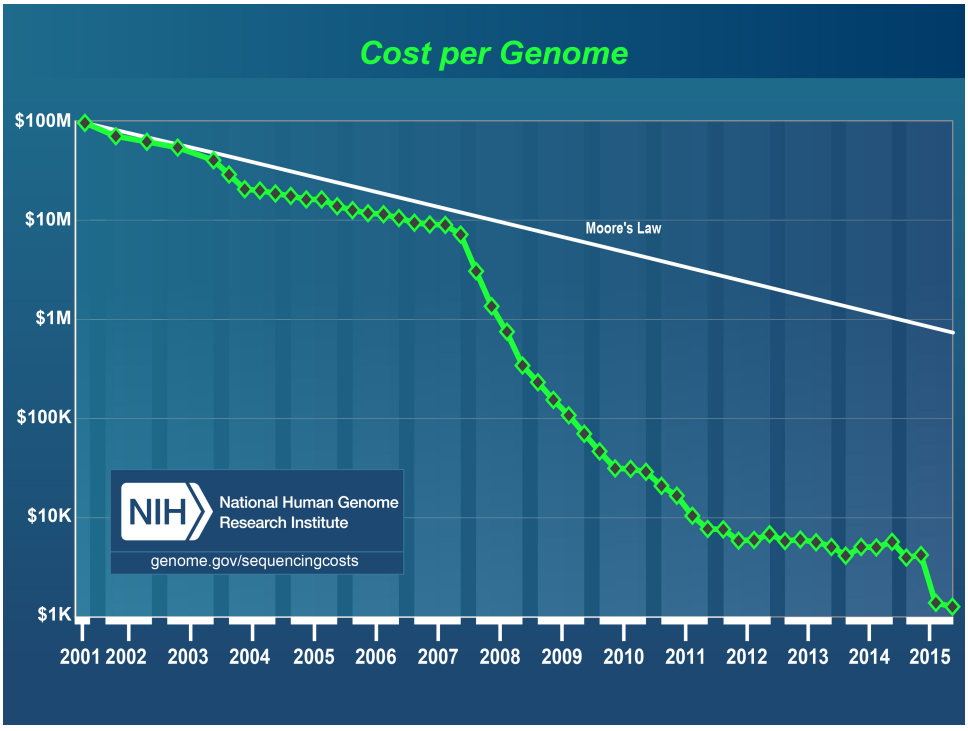

Figure 2.1: Cost of DNA sequencing[136]

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[24], and the large scale sequencing of the Icelandic population[58].

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[137]. The economic cost of cancer to the European Union has been estimated at 126 billion euro in 2009[84], and in the US $124.5 billion USD in 2010[151]. Because of the genetic nature of the disease studying genomes of cancer patients helps uncover the mechanisms behind the development and evolution of cancer[131].

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[60] . 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[131].

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[116].

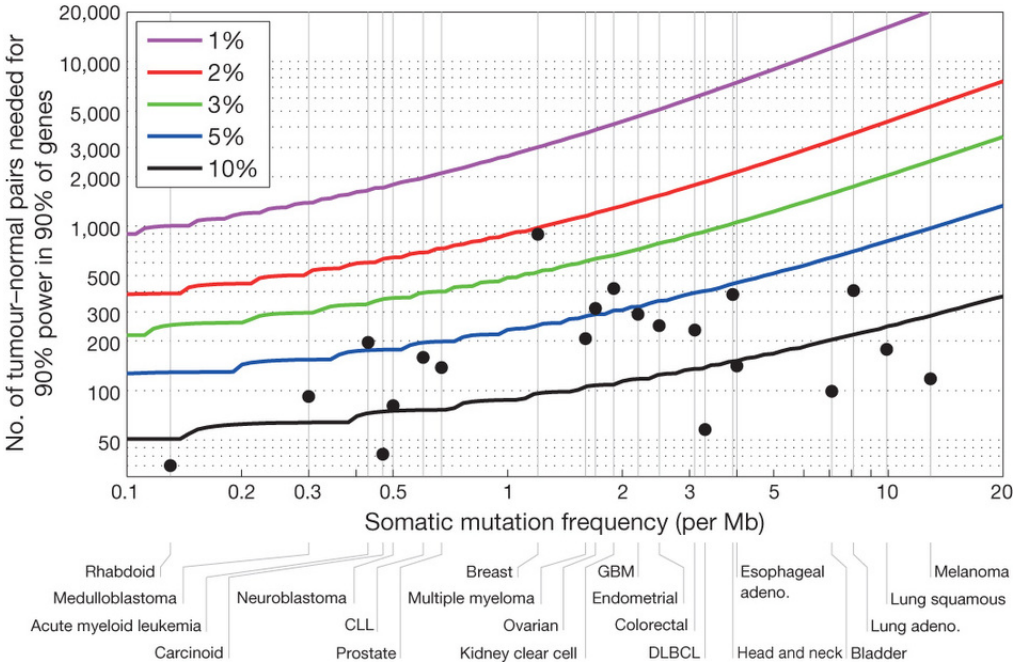

Figure 2.2: Sequencing sample size required by mutation rate[74].

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)[47], 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.

2.1.5 Clinical Genomics

2.2 Computational Methods for Next Generation Sequencing

Abracadabraca. Because the size of a typical genome is millions to billions of base-pairs 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[82] (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

FASTQ - BAM/SAM - CRAM - VCF -

2.2.2 Base-calling

2.2.3 Alignment

BWA/MEM - Bowtie - Gem -

2.2.4 Raw Data QA

2.2.5 Germline SNP Calling

samtools - freebayes - GATK - Platypus

2.2.6 Germline Indel Calling

2.2.7 Germline Structural Variant Calling

Delly - Hydra -

2.2.8 Variant Filtering

2.2.9 Somatic SNP Calling

Mutect - Muse - Pindel -

2.2.10 Somatic Indel Calling

2.2.11 Somatic Structural Variant Calling

2.2.12 Germline Variant Annotation

Annovar - Variant Effect Predictor -

2.2.13 Somatic Variant Annotation

2.2.14 de-novo Assembly

Velvet - Abyss -

2.3 High Performance and High Throughput 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[53]:

> 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[118], where all of the memory was shared between multiple processors, by the late 1980's machines with a distributed memory model[102], 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[61] or LSF[156] 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.

2.4 Cloud Computing

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[19].

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[93].

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 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[71].

To help alleviate these concerns an open-source cloud platform called Openstack was launched in 2010 jointly by Rackspace Hosting and NASA[124]. 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 for the basis for most academic private and community clouds such as EBI Embassy Cloud[27], University of Chicago Open Science Data Cloud[57], Cancer Genome Collaboratory[152], and Helix Nebula[89]. Because these clouds implement the security measures necessary when handling patient data they are a system of choice for large scale bioinformatics analyses.

2.5 Workflow Systems

The focus on workflow stems 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"[134]. One of the key techniques that were devised at the time and served as the prototype for future workflows were "time and motion studies"[14] 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 WMC in 1995 is The Workflow Reference Model[62]. 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[126]. 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[111] 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[37]. 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[83] - A Java-based WfMS built on top of the Ptolemy II[34] execution engine.

Taverna[105] - A Java-based WfMS originally built by myGrid, currently under incubation at Apache Software Foundation.

Galaxy[54] - A Python-based WfMS developed specifically for bioinformatics applications with a focus on GUI-driven development of workflows.

Curcin et al[29] 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[6].

2.6 Service Oriented Architectures

2.7 Stream-based Systems

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[45] (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[122]. 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[146], 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[94]. 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.

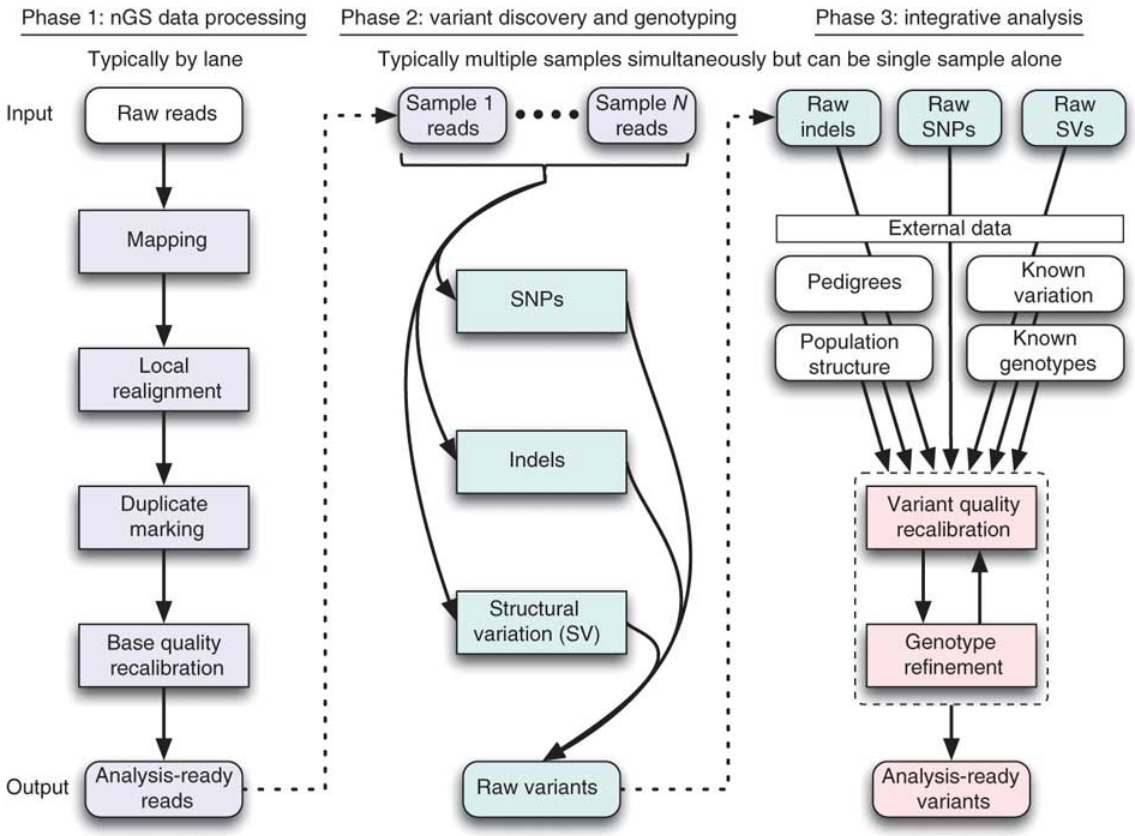

Figure 3.1: NGS Workflow[36].

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[97], or Stochastic Gradient Descent[16] 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 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 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[135] (https://github.com/hashicorp/terraform) - for Cluster Lifecycle Management
- Hashicorp Consul[30] (https://github.com/hashicorp/consul) - for Service Discovery and Service Health Checking
- Saltstack[119] (https://github.com/saltstack/salt) - for Cluster Configuration Management
- Apache Airflow[11] (https://github.com/apache/incubator-airflow) - for Workflow Management
- RabbitMQ[113] (https://github.com/rabbitmq/rabbitmq-server- for Queuing
- Celery[64] (https://github.com/celery/celery) - for Task Scheduling
- Collectd[128] (https://github.com/collectd/collectd) - for Metrics Collection
- InfluxData InfluxDB[66] (https://github.com/influxdata/influxdb) - for Metrics Storage
- Grafana[104] (https://github.com/grafana/grafana/) - for Metrics Dashboards
- Logstash[127] (https://github.com/elastic/logstash) - for Log Harvesting
- Elasticsearch[13] (https://github.com/elastic/elasticsearch) - for Log Indexing and Aggregation
- Kibana[70] (https://github.com/elastic/kibana) - for Log Even 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 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:

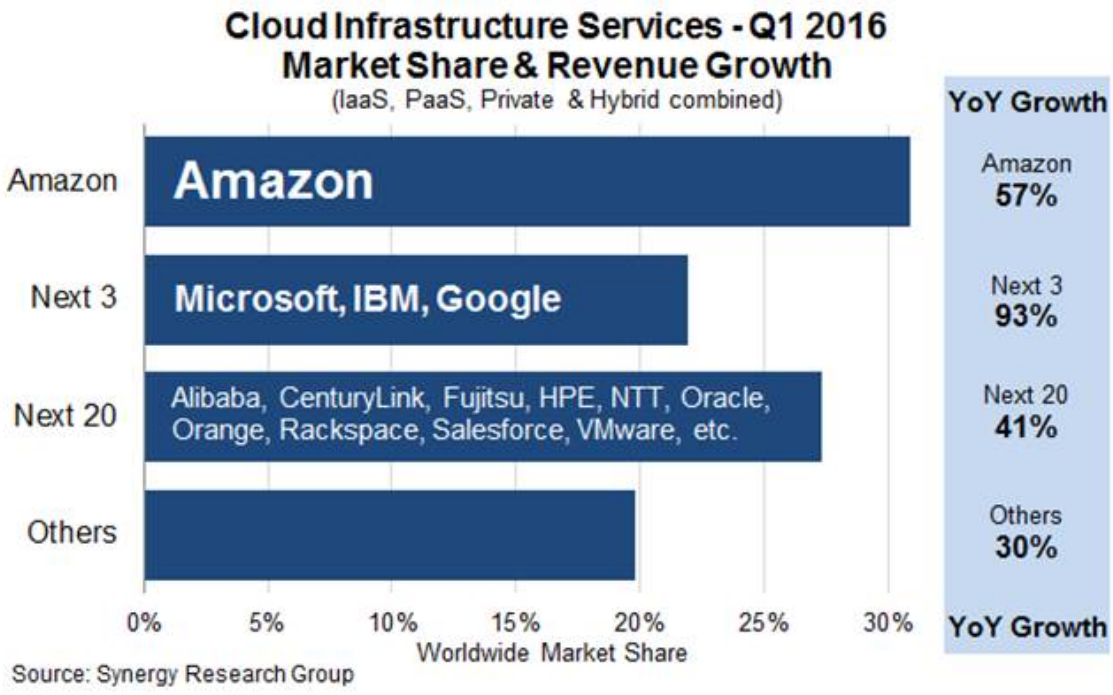

Figure 3.2: Cloud market share by vendor

- The cloud computing market segment enjoys a great deal of growth and significant shifts in growth year-over-year as evidenced by Figure3.2, 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 use 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 13 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.ind`

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 14 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 carriers 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 `package.installed` and

```
install_server:
  pkg.installed:
    - name: postgresql95-server.x86_64

initialize_db:
  cmd.run:
    - name: /usr/pgsql-9.5/bin/postgresql95-setup initdb
    - unless: stat /var/lib/psql/9.5/data/postgresql.conf

/var/lib/pgsql/9.5/data/postgresql.conf:
  file.managed:
    - source: salt://postgres/config/postgresql.conf
    - user: postgres
    - group: postgres
    - mode: 600
    - makedirs: True


start_server:
  service.running:
    - name: postgresql-9.5
    - watch:
      - file: /var/lib/pgsql/9.5/data/*
```

. 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.

```
test_data_sample_path: /shared/data/samples

test_data_base_url: http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/

test_samples:
  NA12874:
    -
      - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam
      - 88a7a346f0db1d3c14e0a300523d0243
    -
      - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam.bai
      - e61c0668bbaacdea2c66833f9e312bbb
```

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.

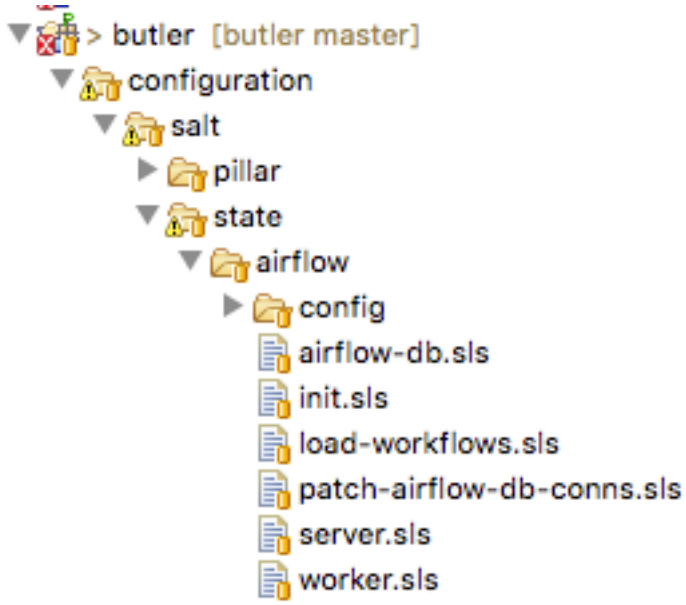

Figure 3.7: Salt state files for Airflow Workflow Engine

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

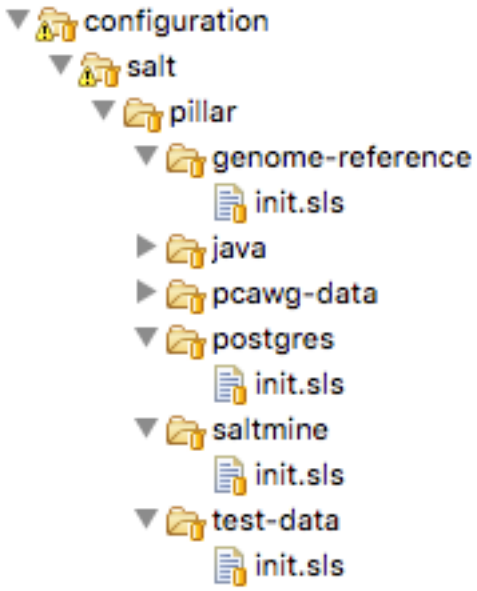

Figure 3.8: A set of Salt Pillar definitions

Listing 15 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 16 demonstrates how the Jinja templating engine is used to look up the IP Address of servers in the cluster that have the `consul-bootstrap`

```
base:
  '*':
    - consul
    - dnsmasq
    - collectd
  'G@roles:monitoring-server':
    - influxdb
    - grafana
  'G@roles:job-queue':
    - rabbitmq
```

or 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 17 demonstrates how the State mapping to Roles is accomplished in a Top File. Based on this Top File all VMs will get the `monitoring-server` states. VMs with the `influxdb, and grafana` role will get `job-queue`, and VMs with the `rabbitmq` role will get the `"salt target_expression command_expression"` 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` where:
`target_expression` is the name of the Salt CLI.
`command_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.
`state.apply` is an expression that determines what actual command to run on the targeted VMs. The `state.apply` can be as simple as running a shell command on the target VMs, or it can apply a particular named state via the `state.highstate` command, or it can apply all matching states via the special `salt -G 'roles:worker' state.apply airflow.patch-airflow-db-conns` command.
For example, `airflow.patch-airflow-db-conns` applies the `worker` state to all

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

VMs that have the 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.

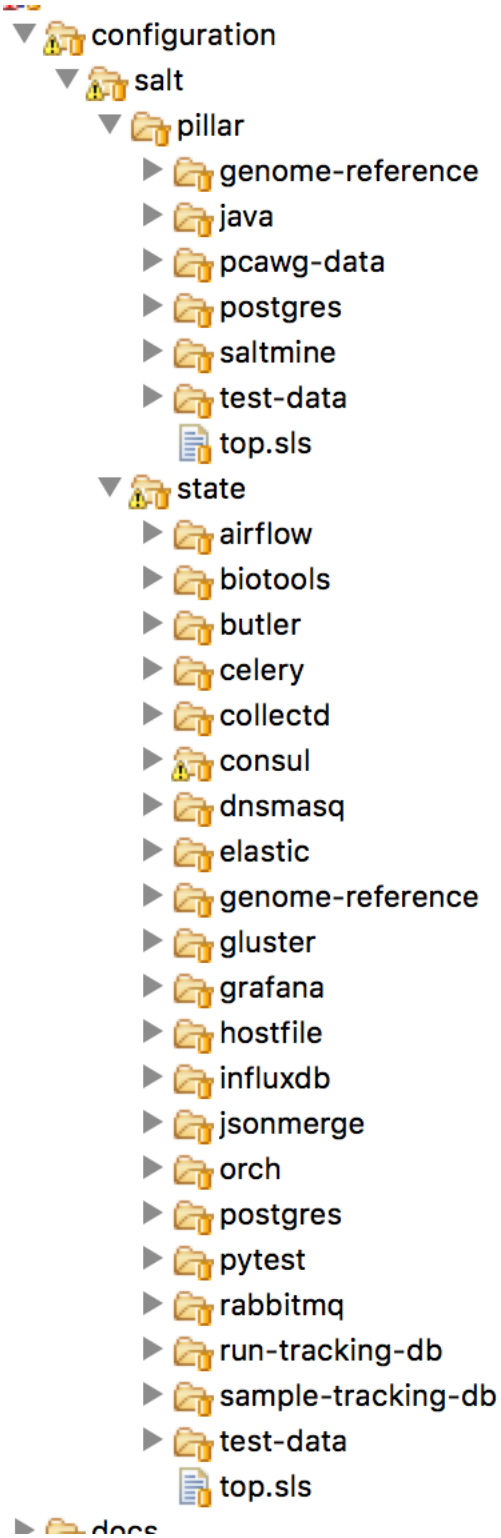

Figure 3.9: Salt States and Pillar used in Butler

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:

`remote-exec`

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 `highstate` provisioner. Salt's

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

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.

`db-server`

The Top File mapping of States to the `tracker` 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 `"salt target_expression command_expression"` state mapped to it in the Top File, to install the RabbitMQ queueing system.

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

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

```
'G@roles:worker':
  - dnsmasq.gnos
  - celery
  - airflow
  - airflow.load-workflows
  - airflow.worker
  - 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 role which simply allows such VMs to run workflows in principle by installing the necessary components of the workflow engine and Butler Analysis Tracker.

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

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

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:

```
highstate
```

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 `class airflow.models.DAG(dag_id, schedule_interval=datetime.timedelta(1), start_da` 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

`BaseOperator`

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 `class airflow.models.BaseOperator(task_id, owner='airflo` class.

`PythonOperator`

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 `task_2.set_upstream(task_1)` 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_1.set_downstream(task_1)` or `set_downstream()`. The `set_upstream()` and `setup.py` 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:

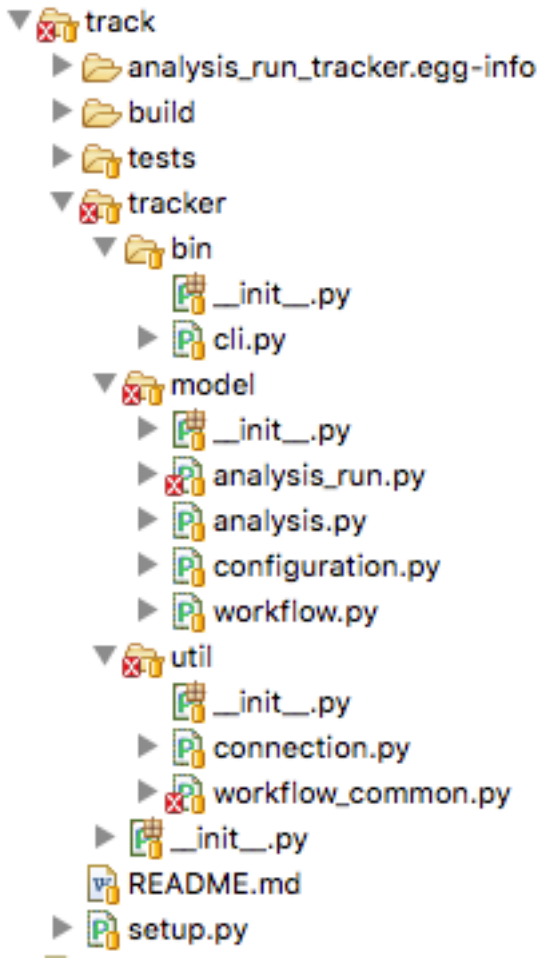

Figure 3.10: File hierarchy of the tracker module

At the root of the hierarchy are the README file and the module installation script `bin`. Inside the `cli.py` directory is the Analysis Tracker CLI implementation - `model`. Inside the `util` 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 `connection.py` directory are utility functions - `workflow_common.py` for connecting to the Run Tracking Database, and `create_workflow(workflow_name, workflow_version, config_id)` for implementations of common workflow tasks.
The overall model can be seen in Figure 3.11

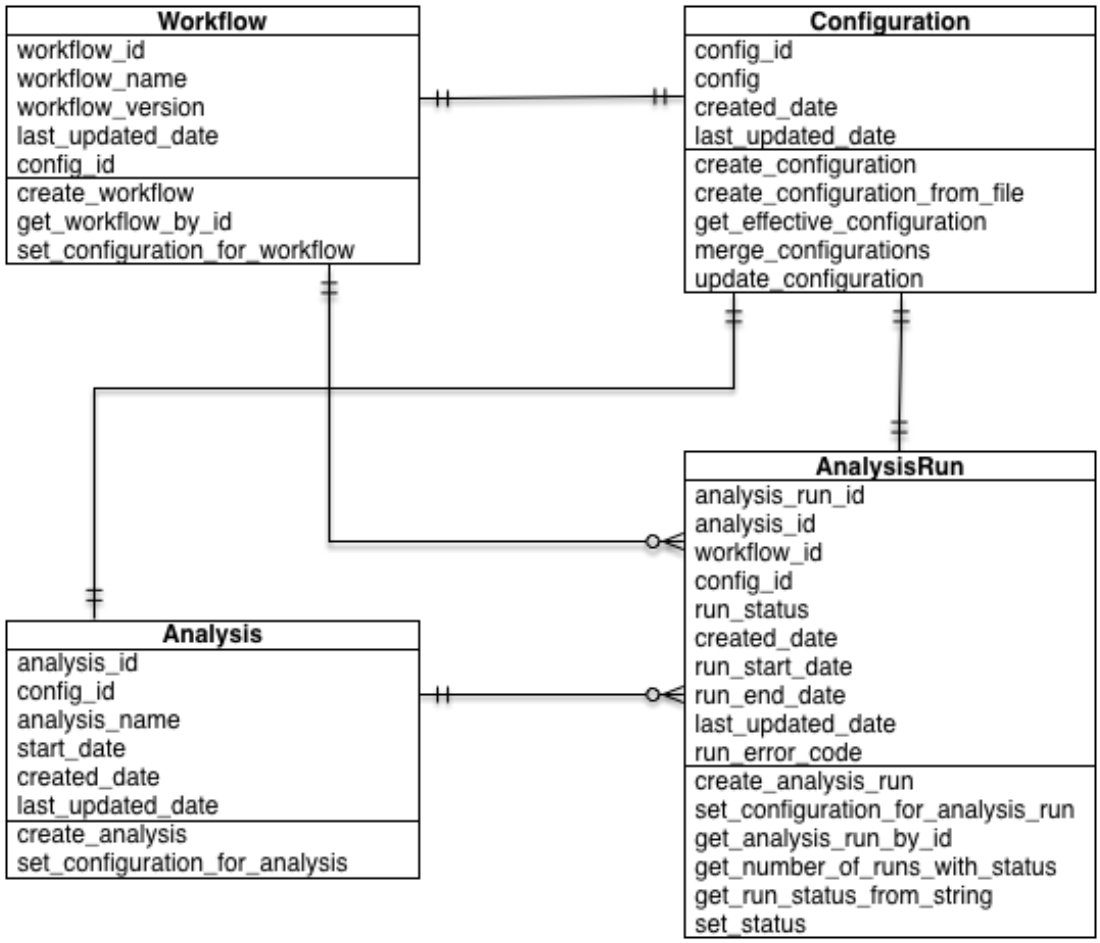

Figure 3.11: UML model of the Analysis Tracker.

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:

`get_workflow_by_id(workflow_id)` - Create a new workflow object with given name, version, and configuration.

`workflow_id` - Retrieve the workflow with ID `set_configuration_for_workflow(workflow_id, config_id)` from persistent storage.

`config_id` - Update the workflow configuration to configuration with ID `create_analysis(analysis_name, start_date, 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:

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

`RUN_STATUS_READY, RUN_STATUS_SCHEDULED, RUN_STATUS_IN_PROGRESS, RUN_STATUS_COMPL` - Update the Analysis configuration to configuration with ID `create_analysis(analysis_name, start_date, 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 `get_run_status_from_string(run_status_string)`.

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:

`create_analysis_run(analysis_id, config_id, workflow_id)` - Translate string-based run statuses into int-based ones.

`set_configuration_for_analysis_run(analysis_run_id, config_id)` - Create an AnalysisRun and store in the database.

`get_analysis_run_by_id(analysis_run_id)` - Update the AnalysisRun configuration to configuration with ID `create_analysis(analysis_name, start_date, config_id)`.

`analysis_run_id` - Get the Analysis Run with ID `set_ready(my_run)`.

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

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

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

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

`RUN_STATUS_ERROR` - Set the status of a given analysis run to `get_number_of_runs_with_status(analysis_id, run_status)`.

`AnalysisRun` - Returns the number of AnalysisRuns of a given status.

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

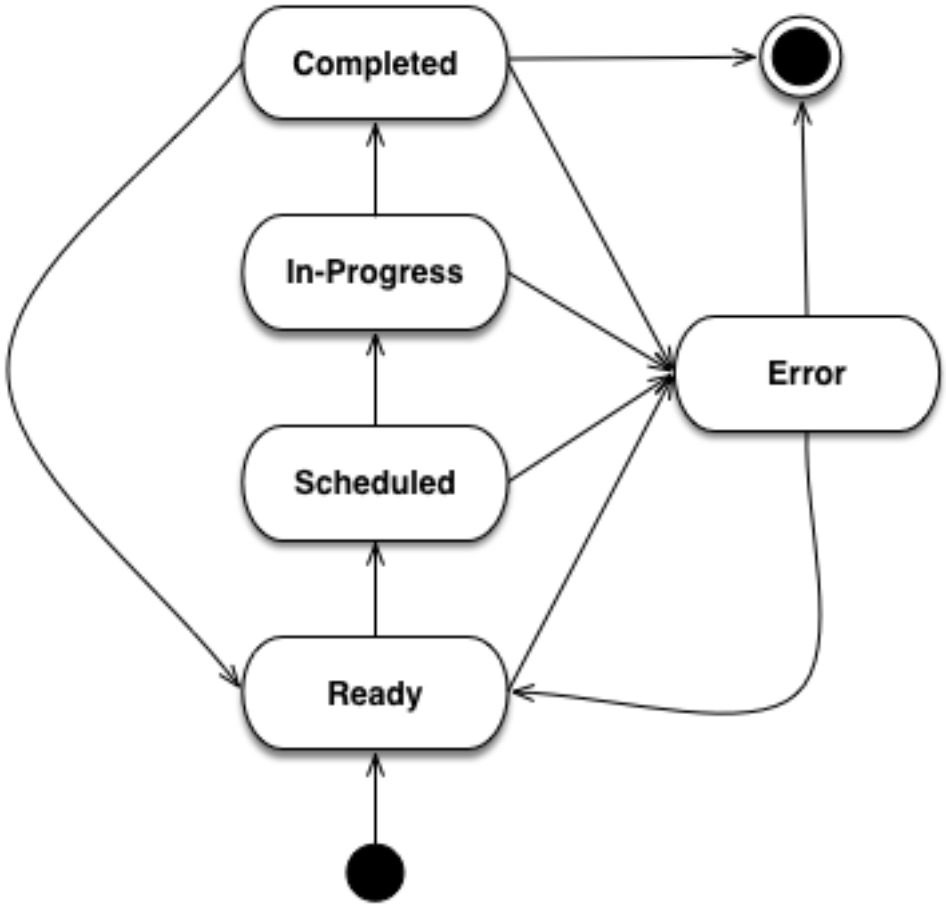

Figure 3.12: State Diagram of Analysis Run Status Transitions.

When an `status` 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_sample(kwargs)` - Get the Configuration supplied to this Workflow.

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

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

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

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

`call_command(command, command_name, cwd=None)` - Test whether the sample files are accessible to the workflow.

`subprocess.call` - Wrapper around Python's `compress_sample(result_filename, config)` method that captures logging information.

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

`UPDATE` - 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 `tracker` 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.

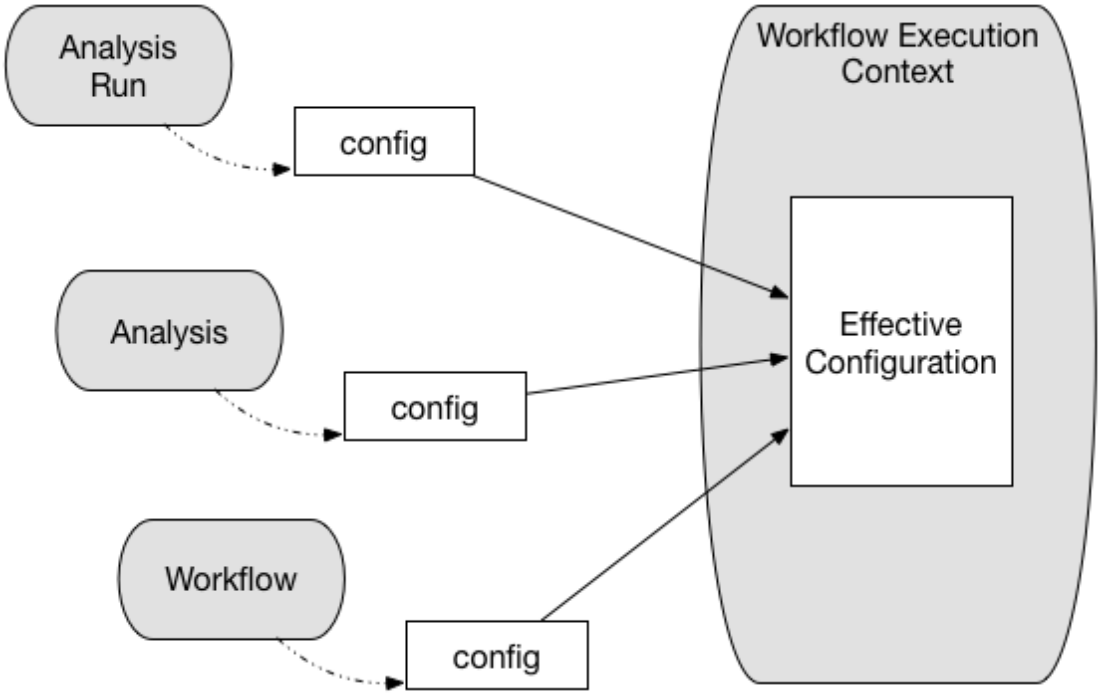

Figure 3.13: Tri-level configuration combines into an effective configuration at runtime.

The configuration facility is built into the `create-workflow` 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-analysis` - creates a new workflow and supports the following parameters:

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

-v, –workfow_version - The version of the workflow.

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

`launch-workflow` - 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.

`update-config` - 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.

`get-run-count` - 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.

`airflow` - 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:

`webserver, scheduler, worker, and flower` - 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 `collectd` 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.

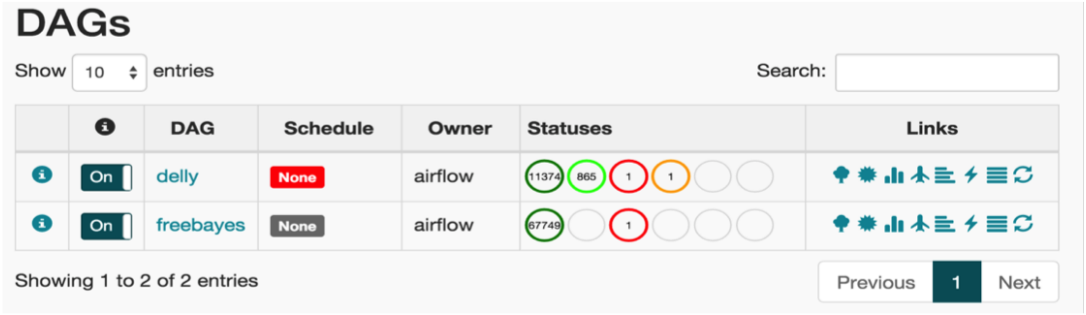

Figure 3.14: The main page of the Airflow Web UI

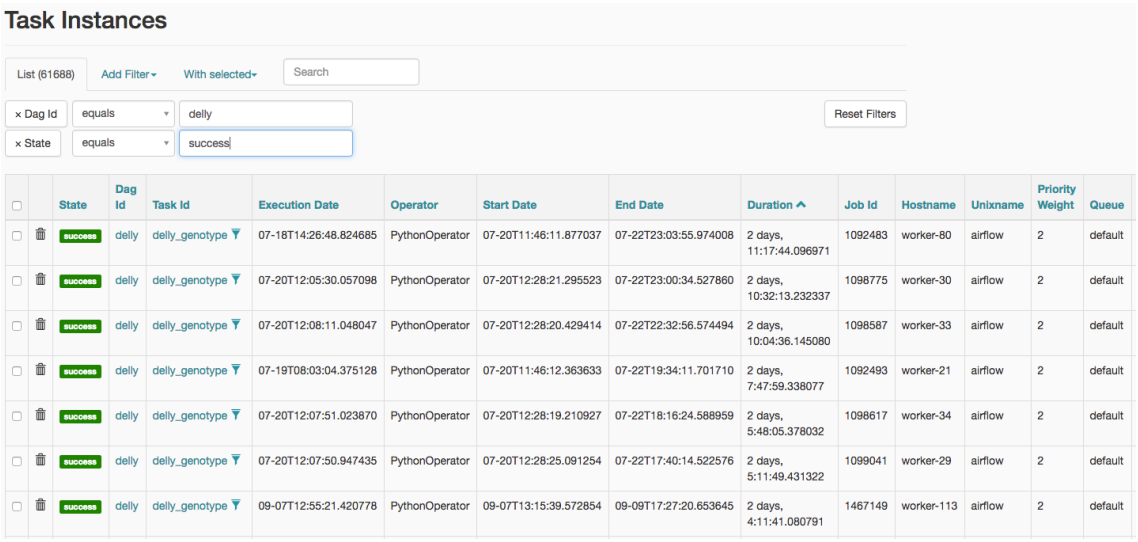

Figure 3.15: Listing of task instances for the freebayes workflow with status of "success".

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

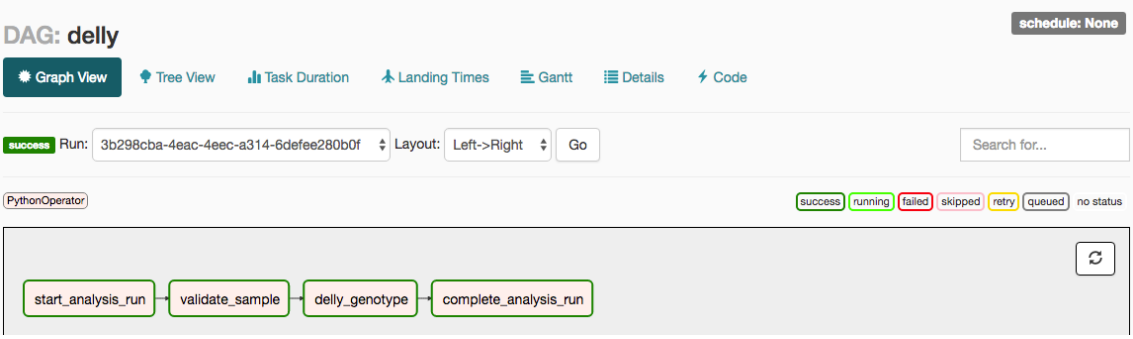

Figure 3.16: Details of the execution of a delly workflow with ID 3b298cba-4eac-4eec-a314-6defee280b0f.

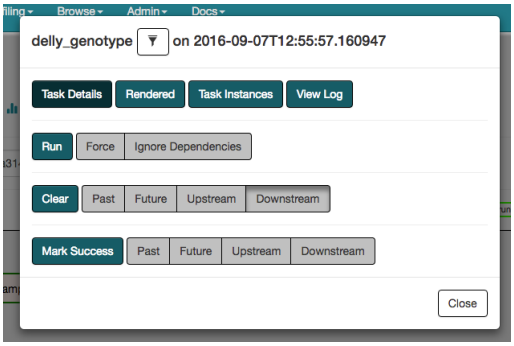

Figure 3.17: A list of actions that can be taken on a workflow task via the Web UI.

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.

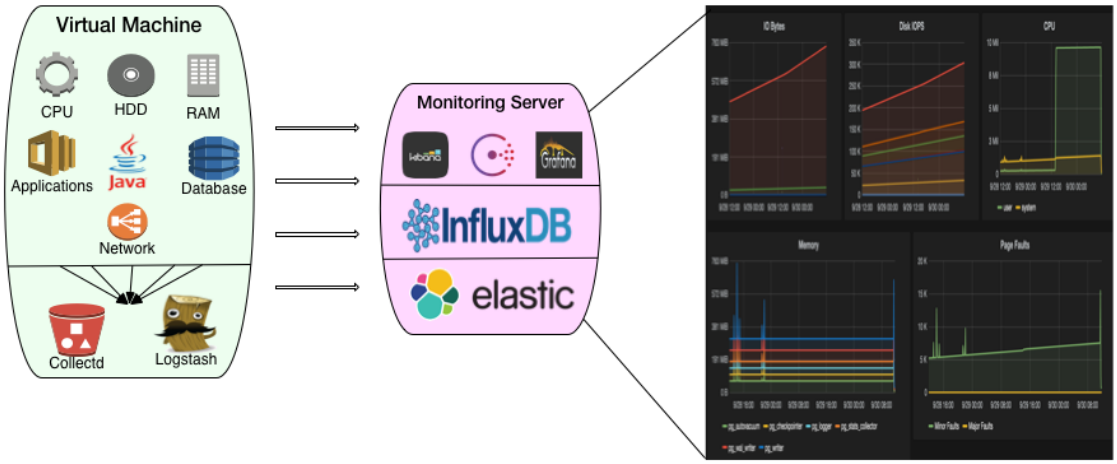

Figure 3.18: Metric monitoring system components

Each VM runs a metric collection daemon called `telegraf`[1] 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 18. 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[66] which is an Open Source database system that is optimized for recording time series data. The configuration for the `influxdb` output plugin in Listing 18 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[106] 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[103] data structure. To the end user the data is organized as a collection of `timestamp`. Each point has a `measurement`, belongs to a `tag_values`, and records a set of `tag_keys` that correspond to a set of `series`. A particular `tag_values` coupled with a retention policy then forms a `memory_value`.
As an example, when tracking RAM, our `tag_values` is called `memory_value`. The set of `series` is:

- host
- type
- type_instance

An example set of `tag_keys` might be:

- worker-145

- memory
- free

The actual `timestamp` corresponding to this combination of `tag_keys` 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[46, 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[104]. 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.

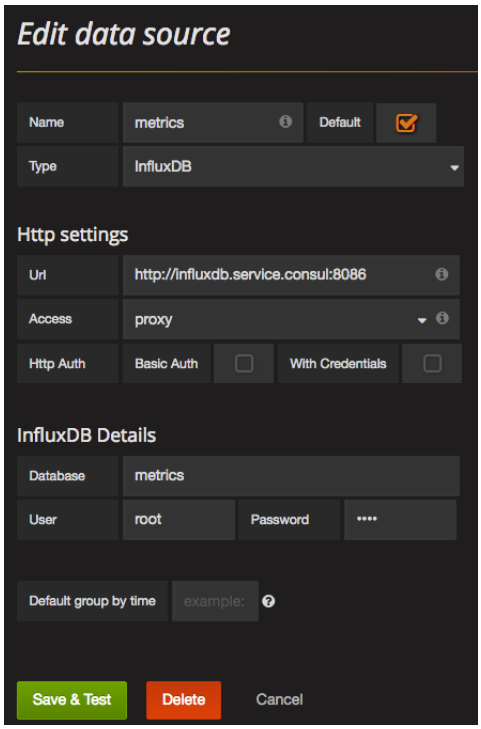

Figure 3.20: InfluxDB Data Source creation for Grafana.

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 dasbhoards 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

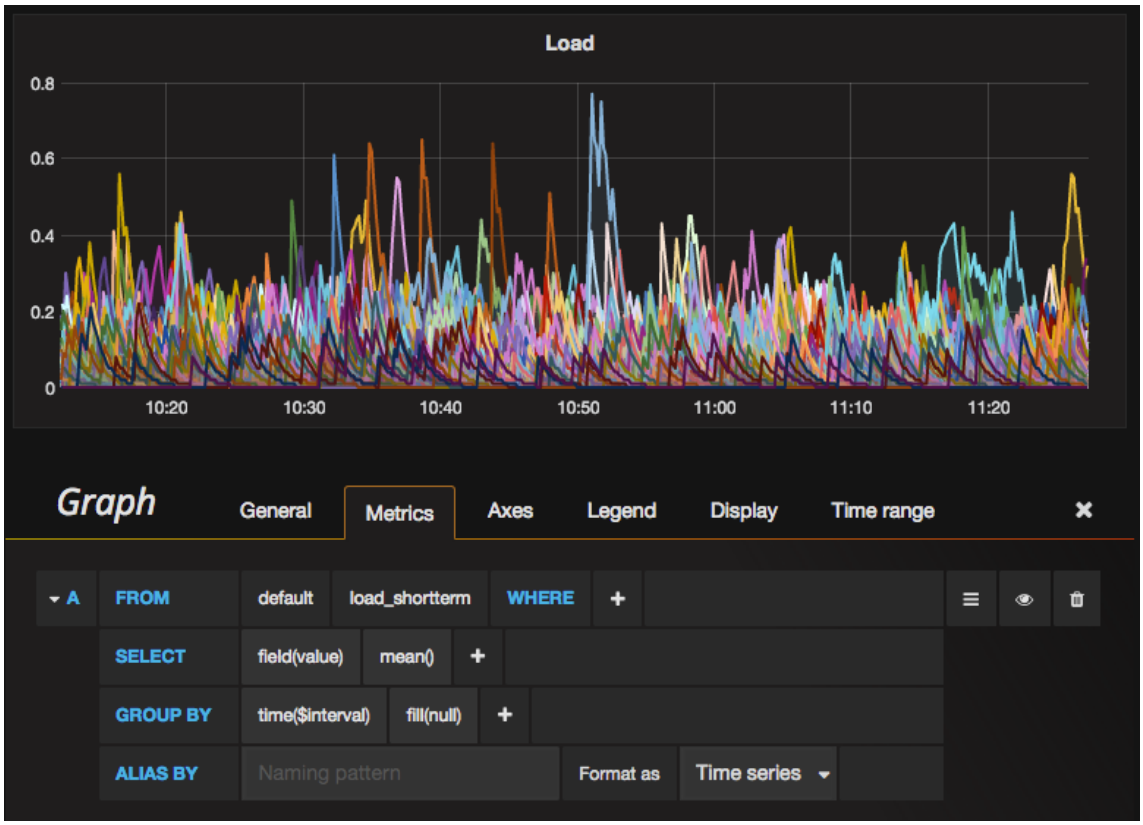

Figure 3.21: Editing a Grafana dashboard panel.

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[13], Logstash[127], and Kibana[70].

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.

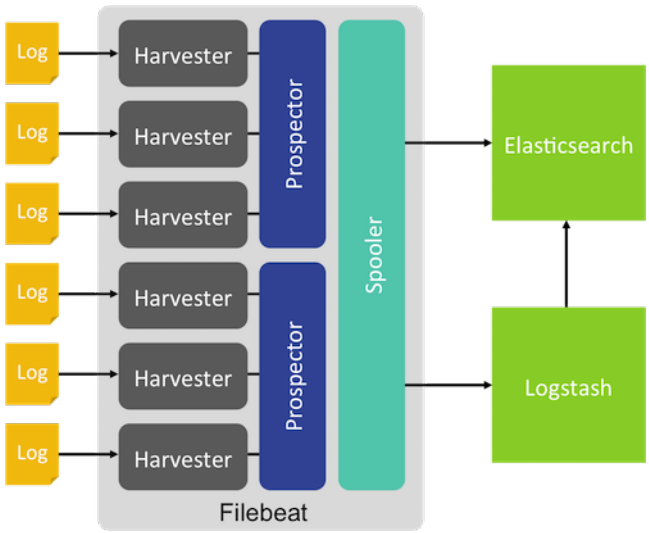

Figure 3.24: Filebeat Architecture (taken from https://www.elastic.co).

As shown in Figure 3.24, Filebeat consits of a set of Prospectors (see Listing 19)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-

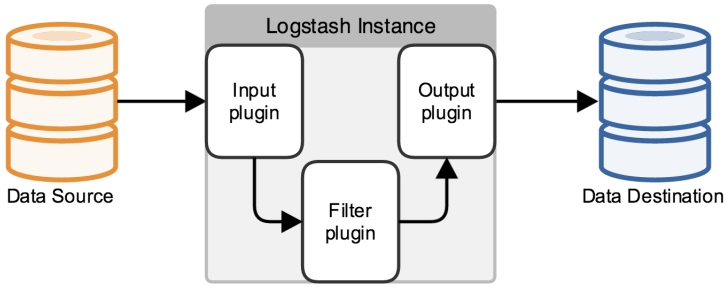

Figure 3.25: A Logstash processing pipeline (taken from https://www.elastic.co).

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.

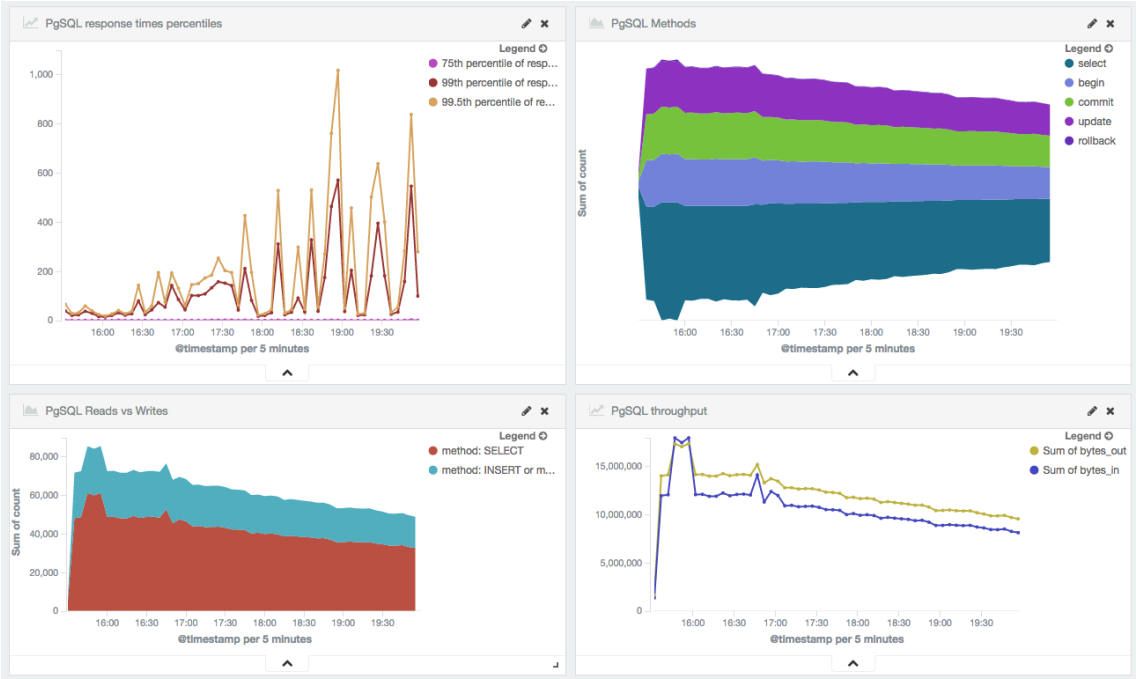

Figure 3.26: Kibana dashboard for PostgreSQL monitoring.

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.

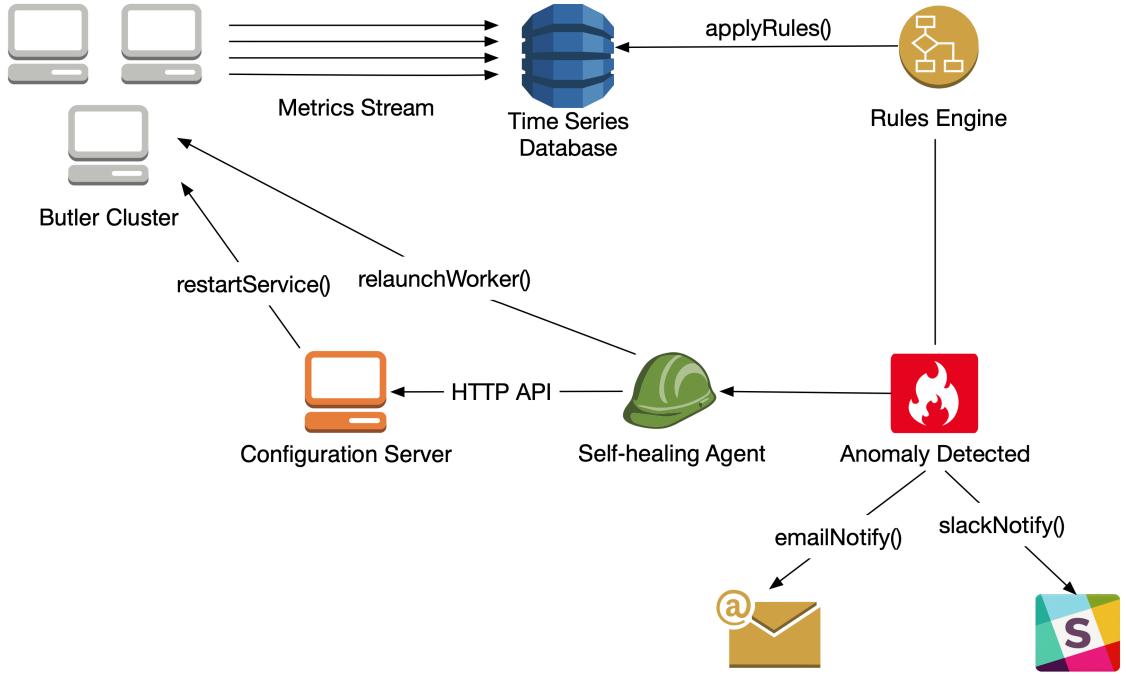

Figure 3.27: The Butler self-healing system.

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.

```
[inputs.procstat]]
exe = "grafana-server"

[[inputs.procstat]]
pid_file = "/var/run/salt-api.pid"

[[inputs.procstat]]
pattern = ".*airflow scheduler.*"

[[inputs.procstat]]
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 `consul-bootstrap`[2] 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:

$$f(\vec{\mathbf{M}}_\mathbf{n}) \quad op \quad C \tag{3.1}$$

given:
$M = \{M_t; t \in T\}$ - a metric time series
$\vec{\mathbf{M}}_\mathbf{n} = [M_{t-n+1} \quad M_{t-n+2} \quad ... \quad M_t]$ - a vector of n most recent observations of M
$f : \mathbb{R}^\ltimes \mapsto \mathbb{R}$ - a function of $\vec{\mathbf{M}}_\mathbf{n}$ 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 Kapacitor[3] 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. Kapacitor 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 20 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 21 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 20 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[4] 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.

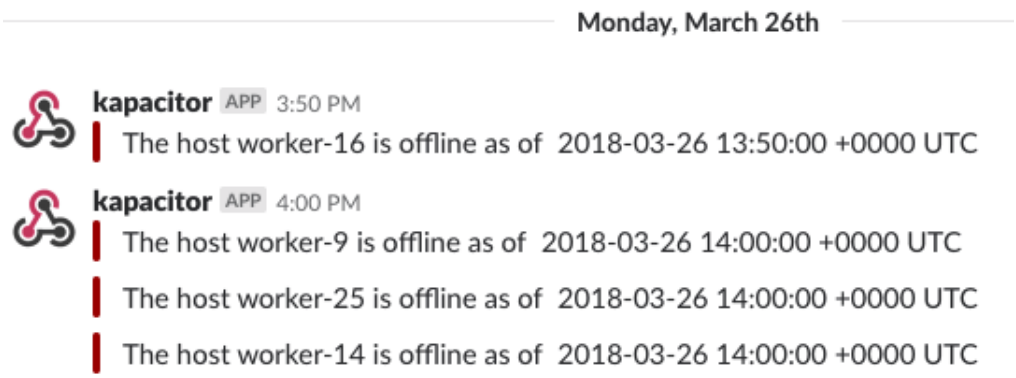

Figure 3.28: Self-healing alerts sent to a Slack channel.

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

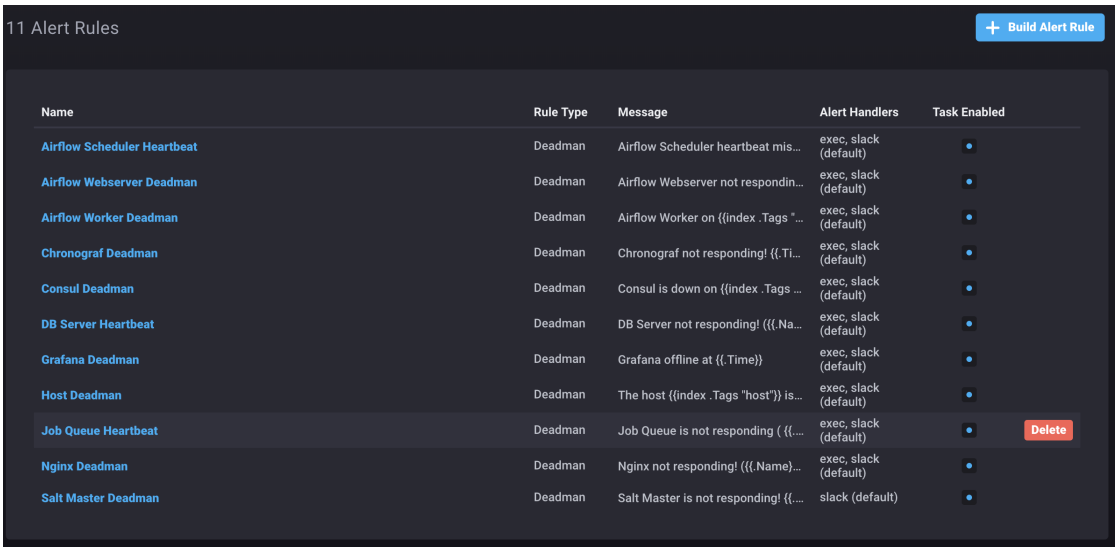

Figure 3.29: List of rule definitions within chronograf.

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[5] 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.

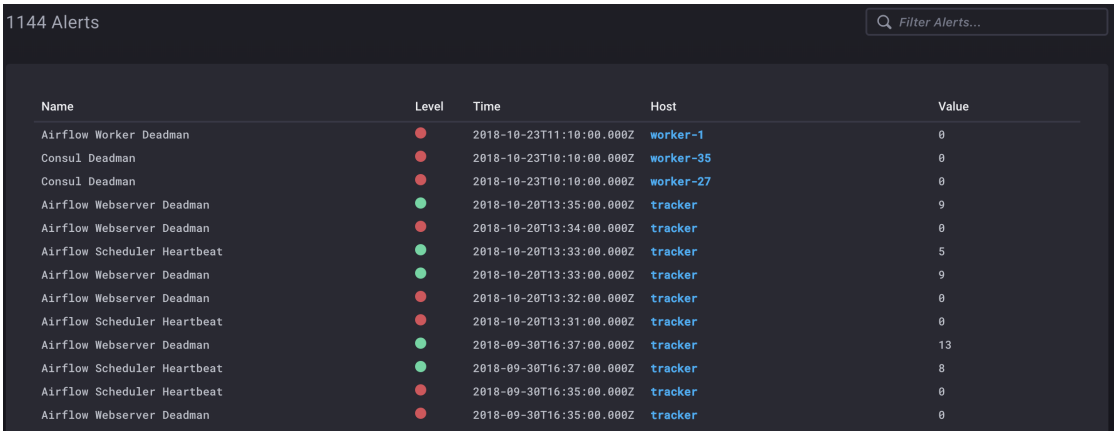

Figure 3.30: Butler alert history.

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 22), 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 kapacitor 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.

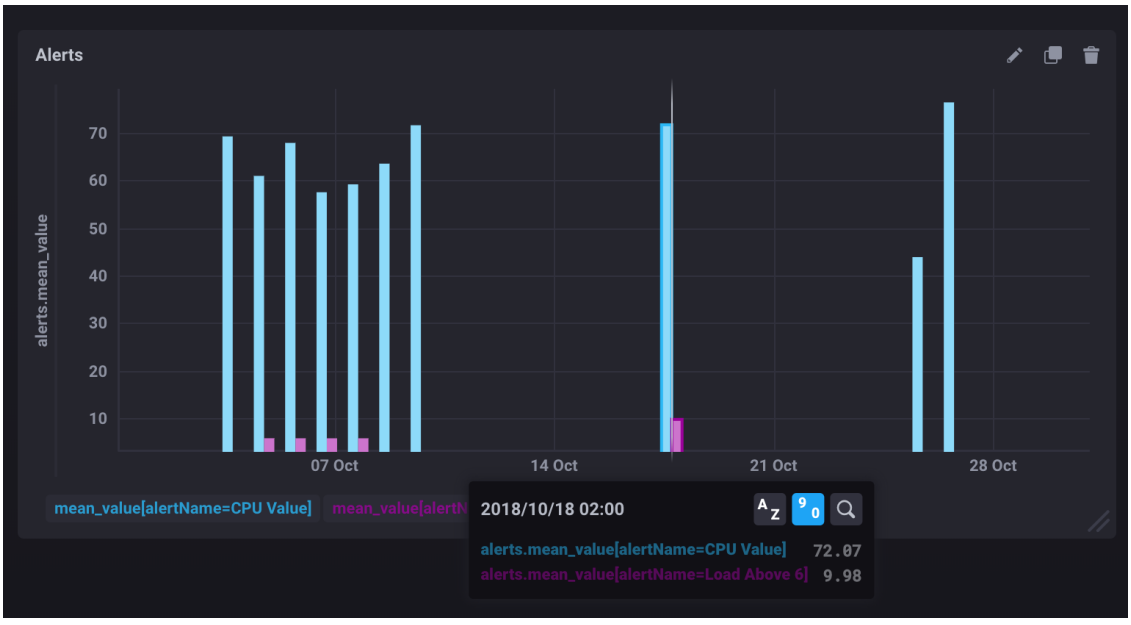

Figure 3.31: Butler timeline of alerts for high CPU utilization.

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.

```
#!yaml|gpg
tf_vars: |
  -----BEGIN PGP MESSAGE-----

  hQEMA0LYc5Mk0CPHAQf8CeN7ykRp518Fm3co0DF5q8pwk9O92ctmDnAhhAascZ2c
  3QESmEIswLVWhKjbZ9tqmmOJeBR+i5gadJjeMStGLHjJm4hNeWLvduk9y63VvhO7
  sjm1N+zzEeYINoj5dkFn9ursggwVP/yk7tlKovvhCO6vwOdIh3UI1S+BzfYb79Sd
  aQGMID1n2LoYCy2y1xXjpTlsYU9uVPhQds2WeFL3Kfhc9B8Q/5i58XdiISVg8ueo
  pg98uOOtIe9BruB6m6tRG6f1W3ZDhjpoBE+DrcBm5k8LZ1khZV70+SjTuoMNOfDl
  K84meNBDoATi5x0FpfNLnQWXLjJkxaxWaZdGaBplhdLAJAF/PvyvMG0BX7XJdCcW
  XGs3BoM+NUJqDgjI5gytmLbPRHA2YWUVNTBmUUw6r9abJyBta1w2RwOFMxEGnszL
  kT0kBPZCAoWiCkgug2G5mnwwP9Wh/CERDCuqObPKHdnES0EMkZ6Bpo7cwX2HMUNj
  5NAYt0gJ6uD1n3zGstQ8Crchj7rkDqottQ/b1JkgFoXKGCYLnv9EuZxaRXz1ad0n
  V7We5/GI+4PmemBLKOHqSdaE/z9sveF1xwY1iUb9hHrHZL2s68nJ4T4VU6VSfVZa
  9W1p4QHic5VDacisq/gWYbSL/ERqKA==
```

```
  =B+ao
  -----END PGP MESSAGE-----

terraform_files: /opt/eosc_pilot/deployment/embassy/
terraform_state: /opt/eosc_pilot/deployment/embassy/terraform.tfstate
terraform_vars: /opt/eosc_pilot/deployment/embassy/ebi_credentials.tfvars
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.

```
{{ pillar['terraform_vars'] }}:
  file.managed:
    - contents_pillar: tf_vars
    - user: root
    - group: root
    - mode: 600
    - 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 23 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 tak 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 difficuluty 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[30].
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

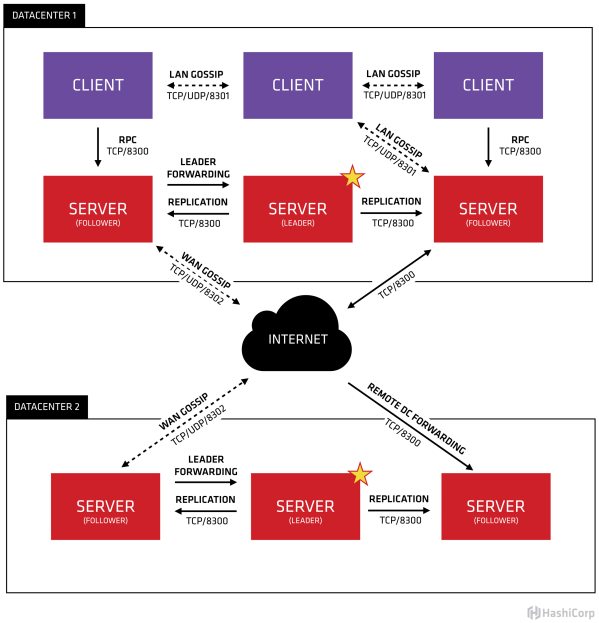

Figure 3.32: Consul high level architecture (from `https://www.consul.io/docs/internals/architecture.html`).

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 24). 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 be available for DNS lookups. The service from Listing 24 will have the name

```
{
        "service": {
                "name": "Postgresql",
                "tags": ["postgresql"],
                "port": 5432}
}
```

, 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[27] 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.

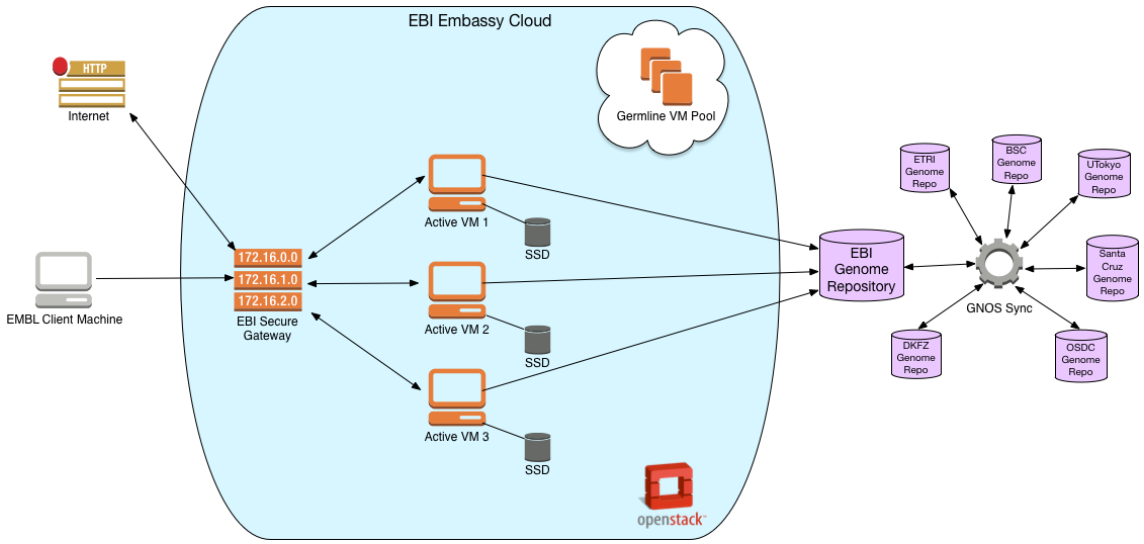

Figure 4.1: Embassy Cloud Architecture

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.

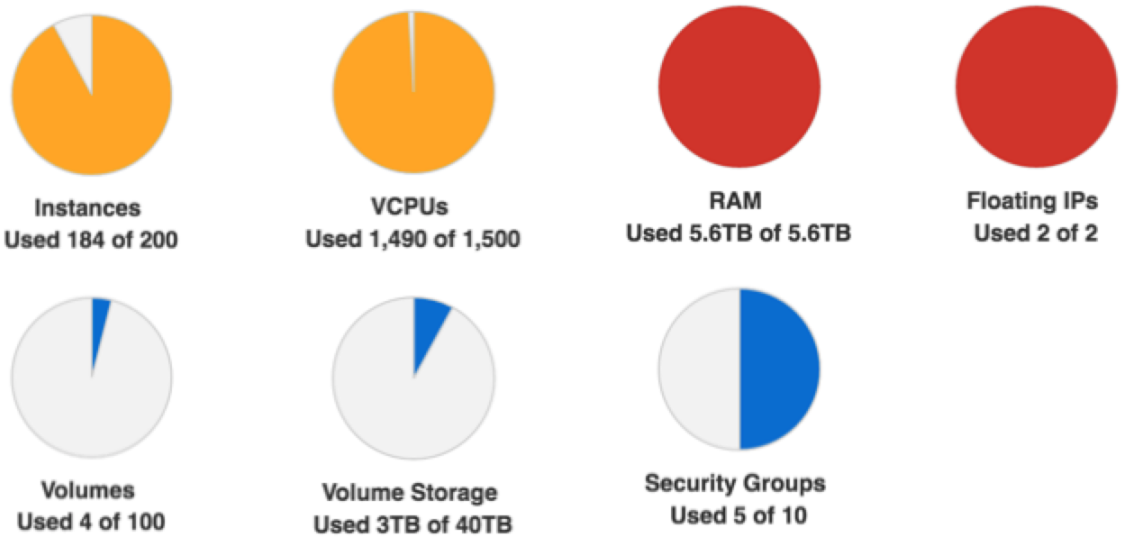

Figure 4.2: Embassy Cloud Dashboard

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.

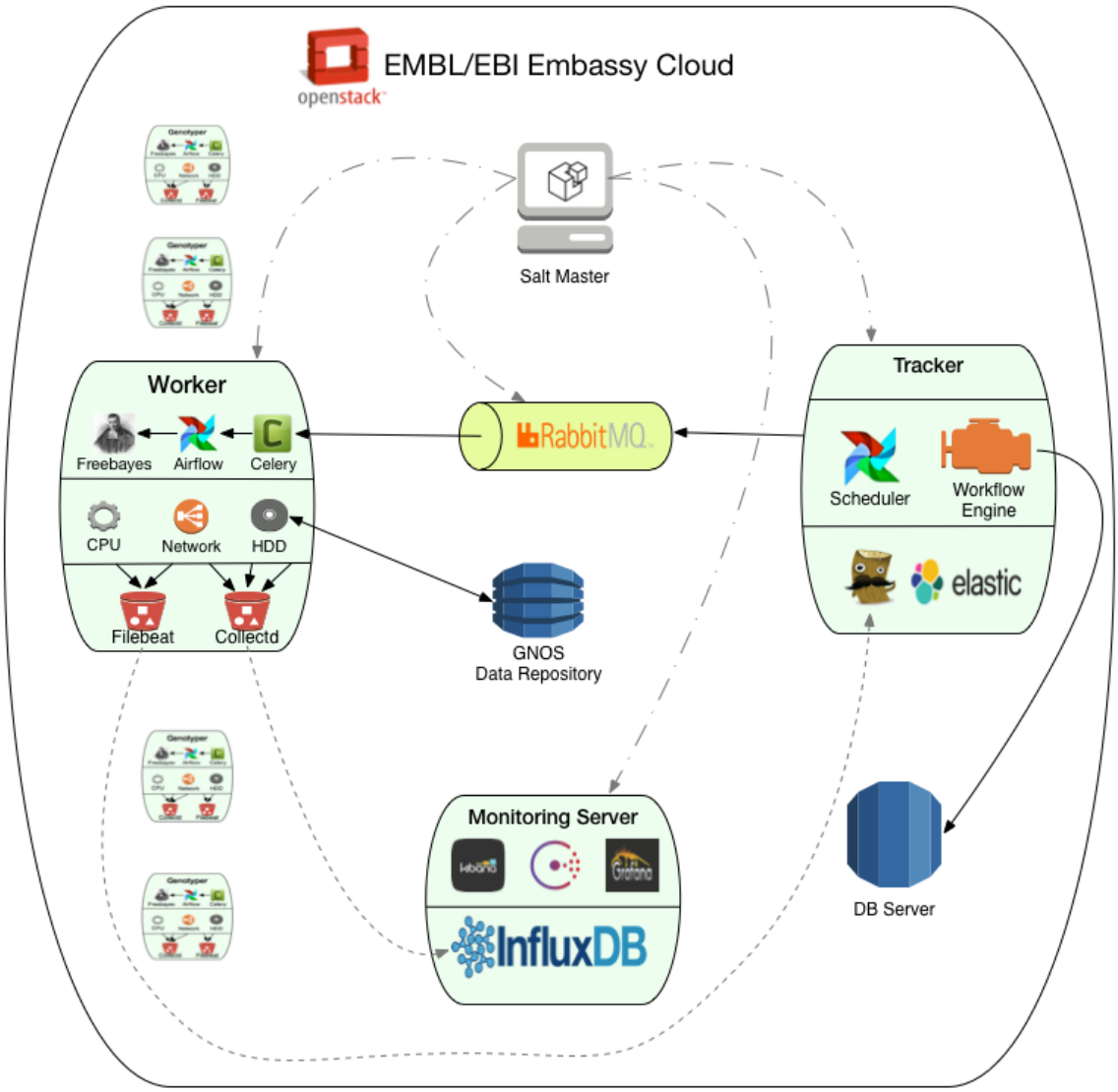

Figure 4.3: Butler Deployment Architecture

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[79]. 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[52], GATK HaplotypeCaller[36], and RTG[22]. 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 25 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          "sample": {
3                  "sample_location": "/gnosdata/tcga/PCAWG.67455c36-aa47-4cc4-8b6d-
                   ↪  9a9012b616ed.bam",
4                  "donor_index": 0,
5                  "sample_id": "f22a72c5-73c8-478d-b03e-04599b9d5321"
6          }
7  }
```

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
                   ↪  --report-genotype-likelihood-max"
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 freebayes
```

Listing 7 provides an example of what an AnaysisRun-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 25 actually invoke a series of commands that perform the bulk of the analysis - calling `bgzip` to generate the discovery call-set, followed by converting the call-set into a binary compressed format (with `tabix`), followed by generating an index file for record-based random access into the binary file (with `rsync`), and followed by an

```
pcawg-germline-clone:
  git.latest:
    - rev: master
    - force_reset: True
    - name: https://github.com/llevar/pcawg-germline.git
    - target: /opt/pcawg-germline
    - submodules: True

/opt/airflow/dags:
  file.symlink:
    - target: /opt/pcawg-germline/workflows/
    - user: airflow
    - group: airflow
    - mode: 755
    - force: True
    - makedirs: True

/tmp/pcawg-germline/scripts:
  file.symlink:
    - target: /opt/pcawg-germline/scripts/
    - user: root
    - group: root
    - mode: 755
    - force: True
    - makedirs: True
```

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 26.

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 min(num_runs, available_runs)
```

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
`python prepare_freebayes_genotyping_config.py create-configs -a 3 -n 150 -t nor`

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:

```
{
        "variants_location": {
            "1":
             ↪  "/freebayes.chr_1.sites.snv_indel.annot.final.vcf.gz",
            "2":
             ↪  "/freebayes.chr_2.sites.snv_indel.annot.final.vcf.gz",
            "3":
             ↪  "/freebayes.chr_3.sites.snv_indel.annot.final.vcf.gz",
            "4":
             ↪  "/freebayes.chr_4.sites.snv_indel.annot.final.vcf.gz",
            "5":
             ↪  "/freebayes.chr_5.sites.snv_indel.annot.final.vcf.gz",
            "6":
             ↪  "/freebayes.chr_6.sites.snv_indel.annot.final.vcf.gz",
            "7":
             ↪  "/freebayes.chr_7.sites.snv_indel.annot.final.vcf.gz",
            "8":
             ↪  "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
            "9":
             ↪  "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
            "10":
             ↪  "/freebayes.chr_10.sites.snv_indel.annot.final.vcf.gz",
            "11":
             ↪  "/freebayes.chr_11.sites.snv_indel.annot.final.vcf.gz",
            "12":
             ↪  "/freebayes.chr_12.sites.snv_indel.annot.final.vcf.gz",
```

```
            "13":
             ↪  "/freebayes.chr_13.sites.snv_indel.annot.final.vcf.gz",
            "14":
             ↪  "/freebayes.chr_14.sites.snv_indel.annot.final.vcf.gz",
            "15":
             ↪  "/freebayes.chr_15.sites.snv_indel.annot.final.vcf.gz",
            "16":
             ↪  "/freebayes.chr_16.sites.snv_indel.annot.final.vcf.gz",
            "17":
             ↪  "/freebayes.chr_17.sites.snv_indel.annot.final.vcf.gz",
            "18":
             ↪  "/freebayes.chr_18.sites.snv_indel.annot.final.vcf.gz",
            "19":
             ↪  "/freebayes.chr_19.sites.snv_indel.annot.final.vcf.gz",
            "20":
             ↪  "/freebayes.chr_20.sites.snv_indel.annot.final.vcf.gz",
            "21":
             ↪  "/freebayes.chr_21.sites.snv_indel.annot.final.vcf.gz",
            "22":
             ↪  "/freebayes.chr_22.sites.snv_indel.annot.final.vcf.gz",
            "X":
             ↪  "/freebayes.chr_X.sites.snv_indel.annot.final.vcf.gz",
            "Y":
             ↪  "/freebayes.chr_Y.sites.snv_indel.annot.final.vcf.gz"
        },
        "results_base_path":
         ↪  "/shared/data/results/regenotype_freebayes_discovery/",
        "results_local_path": "/tmp/regenotype_freebayes_discovery/",
        "freebayes": {
                "mode": "regenotyping",
                "flags": "-l"
        }
}
```

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[143]. 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[101]. 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 27 that we need to provide a list of variant locations that need to be genotyped, split by chromosome, and stored in VCF format[31]. 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.

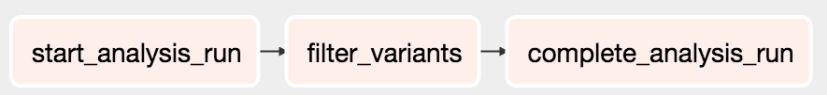

Figure 4.5: Structure of the Butler Variant Filtration workflow

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[31] and vt[133]. vcftools is used for actual variant filtering, while vt is used for variant normalization.

Listing 9: Butler Analysis configuration for VCF filtering.

```
{
        "gnos": {
                "ebi": {
                        "url": "https://gtrepo-ebi.annailabs.com"
                },
                "osdc_icgc": {
                        "url": "https://gtrepo-osdc-icgc.annailabs.com"
                },
                "osdc_tcga": {
                        "url": "https://gtrepo-osdc-tcga.annailabs.com"
                }
        },
        "rsync": {
                "flags": "-a -v --remove-source-files"
        }
}
```

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[35] 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[150] 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.

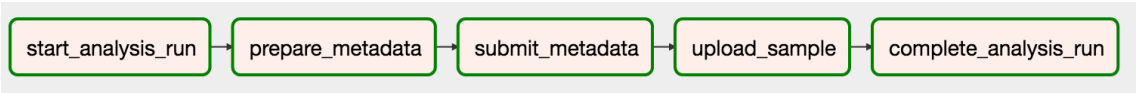

Figure 4.6: Structure of the Butler Data Submission workflow

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 28
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 29 template declaration) and using Python's

```
{
        "gnos": {
                "ebi": {
                        "key_location": "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.p
                },
                "osdc_icgc": {
                        "key_location": "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.p
                },
                "osdc_tcga": {
                        "key_location": "/home/airflow/.ssh/sergei_bionimbus_gnos_ma
                }
        },
        "metadata_template_location": "/opt/pcawg-germline/workflows/gtupload-workfl
        "submission_base_path": "/shared/data/results/freebayes_discovery_gnos_submi
        "destination_repo_mapping": {
                "ICGC": "ebi",
                "TCGA": "osdc_tcga"
        }
}
```

module.

Once the submission is ready, the actual process of submission is carried out using the `destination_repo_mapping` 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 `manifest.xml` dictionary in Listing 29 maintains a mapping between a sample's project and a corresponding GNOS repository name. Listing 28 provides a further mapping between repository names and repository URLs thus allowing `destination_repo_mapping` to submit each sample to its corresponding GNOS repository. The output of the metadata_submit task is a `gtupload` 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

```
def run_delly(**kwargs):

    config = get_config(kwargs)
    sample = get_sample(kwargs)

    sample_id = sample["sample_id"]
    sample_location = sample["sample_location"]

    result_path_prefix = config["results_local_path"] + "/" + sample_id

    if (not os.path.isdir(result_path_prefix)):
        logger.info(
            "Results directory {} not present, creating.".format(result_path_prefi
        os.makedirs(result_path_prefix)

    delly_path = config["delly"]["path"]
    reference_location = config["reference_location"]
    variants_location = config["variants_location"]
    variants_type = config["variants_type"]
    exclude_template_path = config["delly"]["exclude_template_path"]

    result_filename = "{}/{}_{}.bcf".format(
        result_path_prefix, sample_id, variants_type)

    log_filename = "{}/{}_{}.log".format(
        result_path_prefix, sample_id, variants_type)

    delly_command = "{} call -t {} -g {} -v {} -o {} -x {} {} > {}".\
        format(delly_path,
               variants_type,
               reference_location,
               variants_location,
               result_filename,
               exclude_template_path,
               sample_location,
               log_filename)

    call_command(delly_command, "delly")

    copy_result(result_filename, sample_id, config)
```

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[132]. There are various types of structural variants, including:

- Deletions
- Inversions
- Duplications
- Translocations

The Germline Working Group is using a tool called Delly[114] 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 30).
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 31)

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[25] 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.

```
{
        "variants_location":
        ↪  "/shared/data/samples/vcf/delly_deletion_sites/del.sites.bcf",
        "results_base_path":
        ↪  "/shared/data/results/delly_germline_deletions_14_07_2016/",
        "results_local_path": "/tmp/delly_germline_deletions/",
        "variants_type": "DEL"

}
```

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.

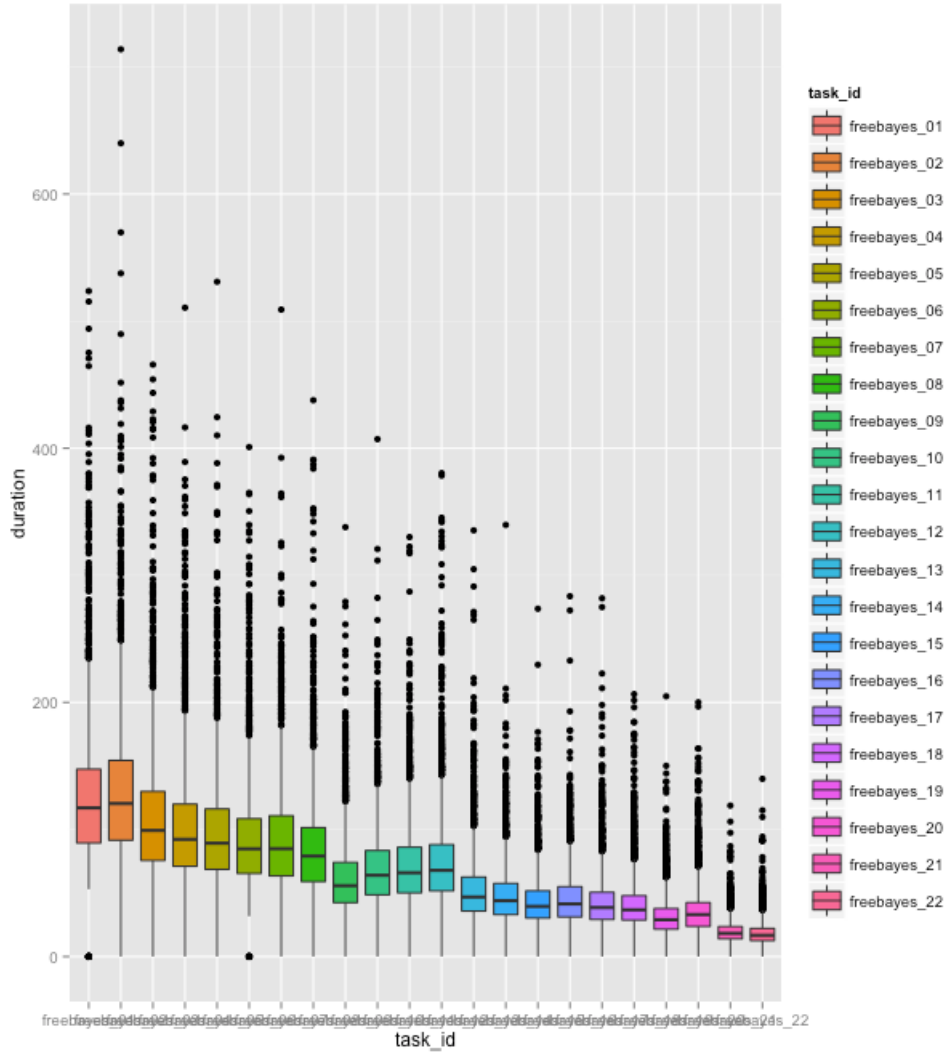

Figure 4.8: Runtimes of freebayes genotyping on the 1% MAF site-list.

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 Discovery

4.3.3 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.

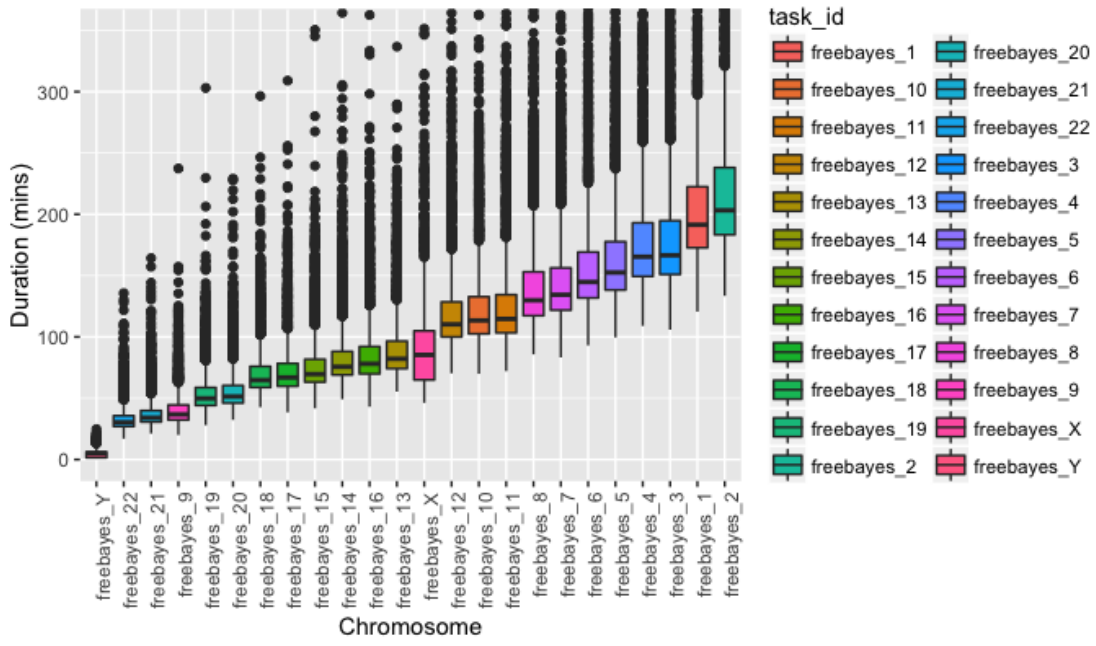

Figure 4.9: Runtimes of freebayes regenotyping on the freebayes discovery call-set.

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.

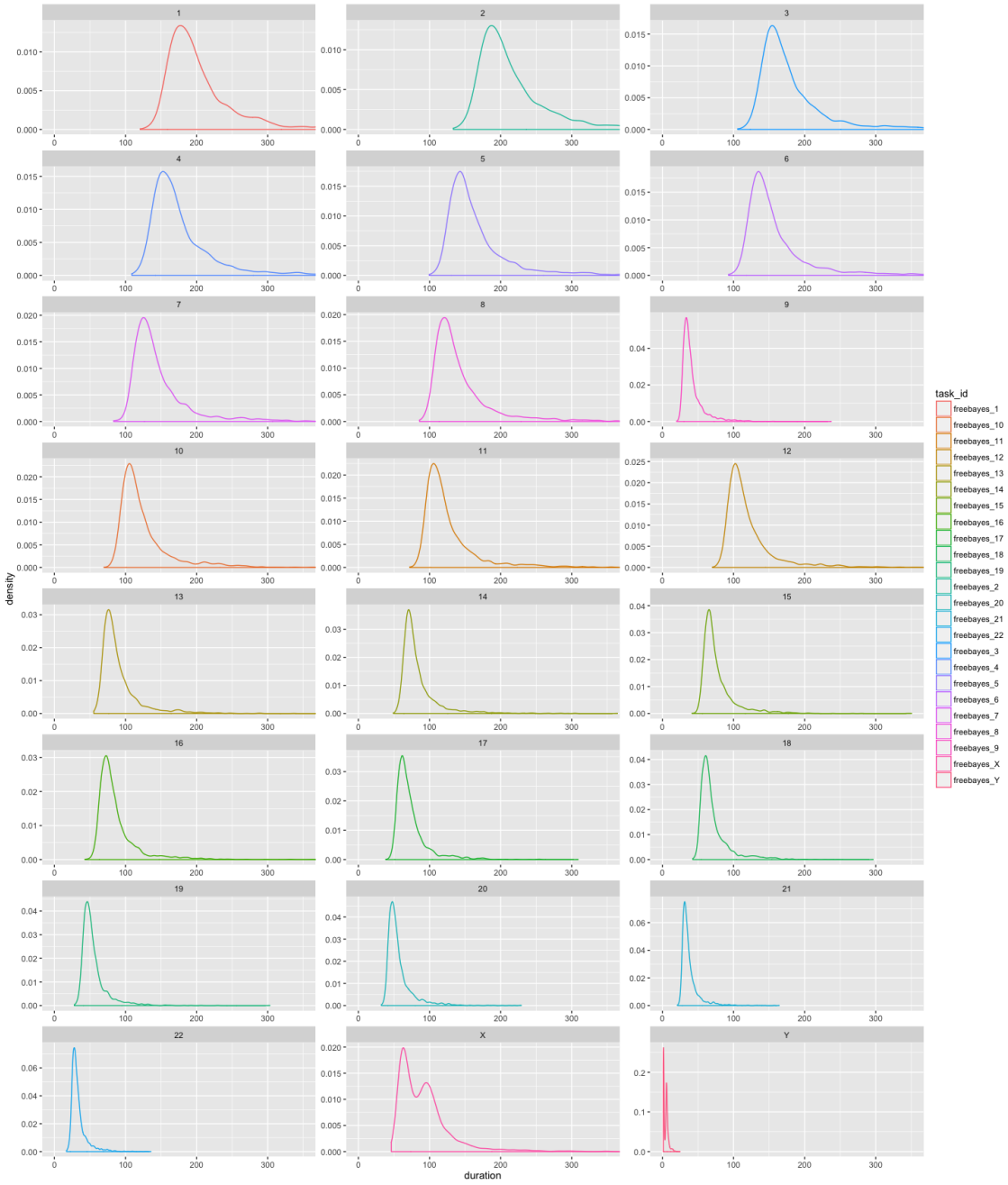

Figure 4.10: Task duration distributions by chromosome.

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.

4.3.4 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.

```
{
        "variants_location":
        ↪  "/shared/data/samples/vcf/delly_deletion_sites/dup.sites.bcf",
        "results_base_path":
        ↪  "/shared/data/results/delly_germline_dups_05_09_2016/",
        "results_local_path": "/tmp/delly_germline_dups/",
        "variants_type": "DUP"

}
```

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.

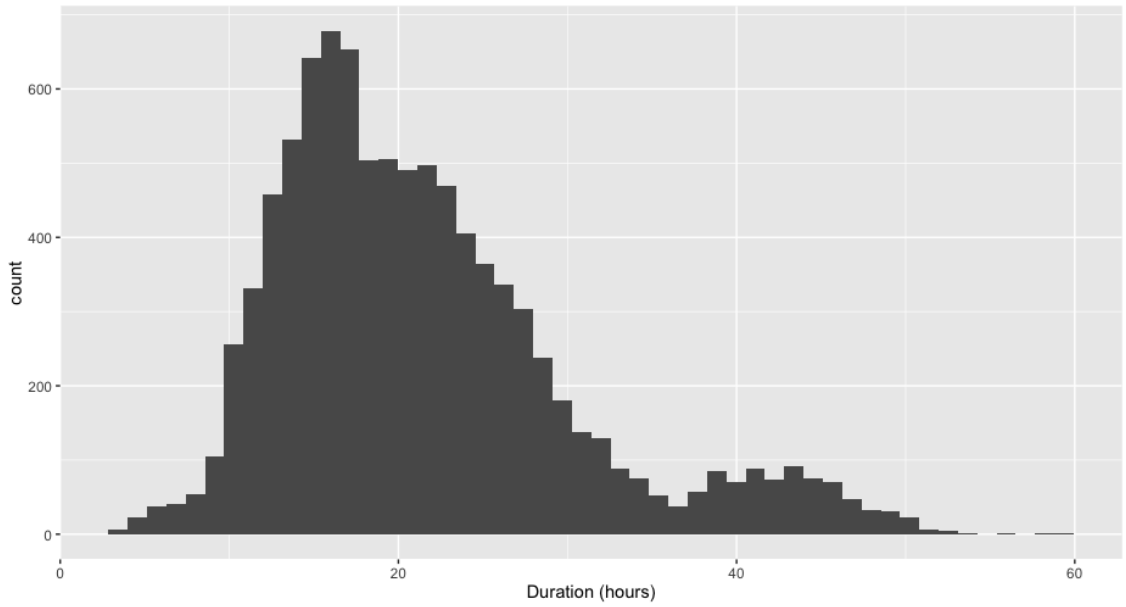

Figure 4.12: Durations of deletion genotyping tasks by sample.

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":
          ↪  "/shared/data/samples/vcf/delly_deletion_sites/dup.sites.bcf",
3         "results_base_path":
          ↪  "/shared/data/results/delly_germline_dups_05_09_2016/",
4         "results_local_path": "/tmp/delly_germline_dups/",
```

```
        "variants_type": "DUP"

}
```

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.

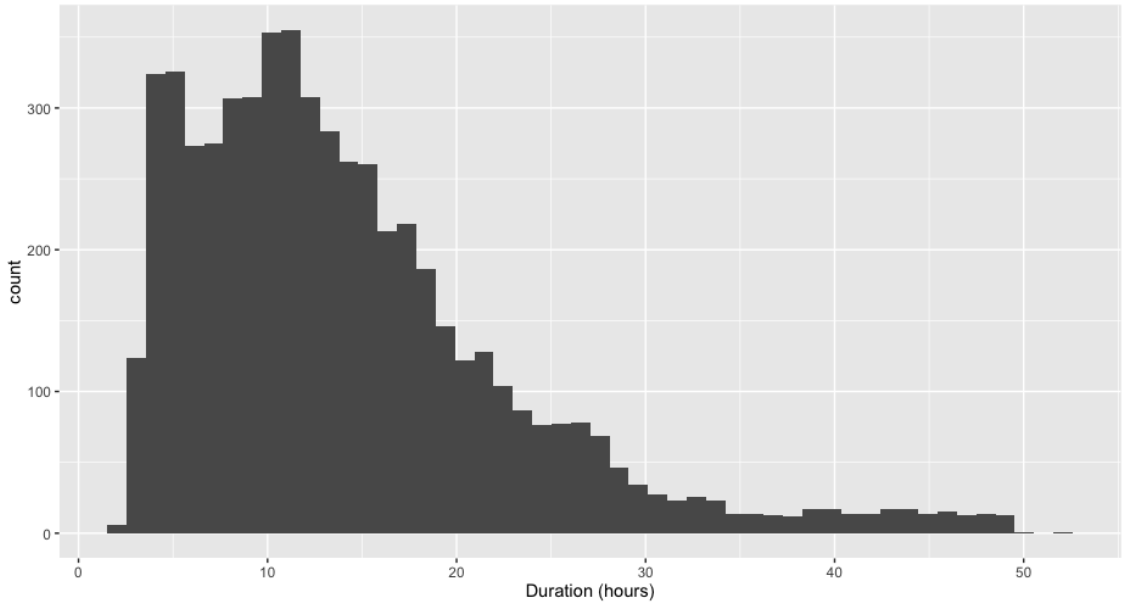

Figure 4.14: Durations of duplication genotyping tasks by sample.

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.

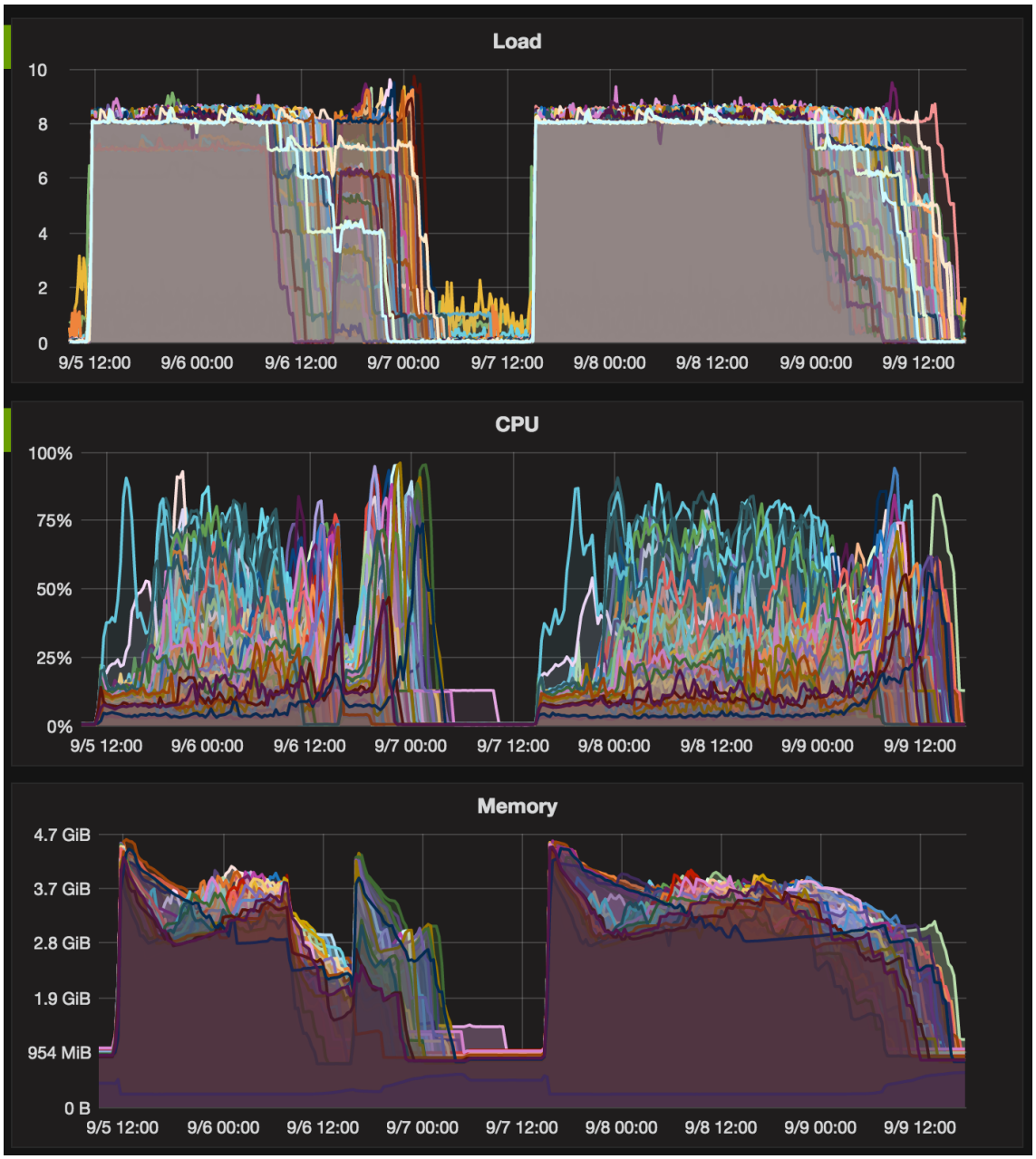

Figure 4.15: Cluster performance during the duplication genotyping analysis.

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.

4.4 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 `create-workflow` 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[77].
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.

4.5 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 wSe 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 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.

4.6 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[90], and the US Precision Medicine Initiative[23] (with up to 1,000,000 genomes). This means that in order to not have to proportionately increase the timeline for theses 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[94] 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[43] initiative that is launching in 2017. Additionally, de.NBI - The German Network for Bioinformatics Infrastructure[63] 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.

Chapter 5

The Rheos Framework

Chapter 6

Discussion and Conclusion

Appendix A

Code Listings

Listing 13: Terraform configuration of a worker VM

```
provider "openstack" {
        user_name = "${var.user_name}"
        password = "${var.password}"
        tenant_name = "${var.tenant_name}"
        auth_url = "${var.auth_url}"
}

resource "openstack_compute_instance_v2" "worker" {
          image_id = "${var.image_id}"
        flavor_name = "s1.massive"
        security_groups = ["internal"]
        name = "${concat("worker-", count.index)}"
        network = {
                uuid = "${var.main_network_id}"
        }
        connection {
                user = "${var.user}"
                 key_file = "${var.key_file}"
                 bastion_key_file = "${var.bastion_key_file}"
                 bastion_host = "${var.bastion_host}"
                 bastion_user = "${var.bastion_user}"
                 agent = "true"

        }
        count = "175"
        key_pair = "${var.key_pair}"
        provisioner "remote-exec" {
                inline = [
                        "sudo mv /home/centos/saltstack.repo
                        ↪  /etc/yum.repos.d/saltstack.repo",
                        "sudo yum install salt-minion -y",
                        "sudo service salt-minion stop",
```

```
32                          "echo 'master: ${var.salt_master_ip}' | sudo tee  -a
                            ↪  /etc/salt/minion",
33                          "echo 'id: ${concat("worker-", count.index)}' | sudo tee
                            ↪  -a /etc/salt/minion",
34                          "echo 'roles: [worker, germline, consul-client]' | sudo
                            ↪  tee -a /etc/salt/grains",
35                          "sudo hostname ${concat("worker-", count.index)}",
36                          "sudo service salt-minion start"
37                  ]
38          }
39  }
```

Listing 14: 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 15: 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      -
```

```
      - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam.bai
      - e61c0668bbaacdea2c66833f9e312bbb
```

Listing 16: Using Salt Mine to look up a server's IP Address.

```
consul-client:
  service.running:
    - enable: True
    - watch:
      - file: /etc/opt/consul.d/*
{%- set servers = salt['mine.get']('roles:(consul-server|consul-bootstrap)',
↪  'network.ip_addrs', 'grain_pcre').values() %}
{%- set node_ip = salt['grains.get']('ip4_interfaces')['eth0'] %}
# Create a list of servers that can be used to join the cluster
{%- set join_server = [] %}
{%- for server in servers if server[0] != node_ip %}
{% do join_server.append(server[0]) %}
{%- endfor %}
join-cluster:
  cmd.run:
    - name: consul join {{ join_server[0] }}
    - watch:
      - service: consul-client
```

Listing 17: Using Top File to map States to Roles.

```
base:
  '*':
    - consul
    - dnsmasq
    - collectd
  'G@roles:monitoring-server':
    - influxdb
    - grafana
  'G@roles:job-queue':
    - rabbitmq
```

Listing 18: Collectd configuration for metrics collection.

```
# Read metrics about cpu usage
[[inputs.cpu]]
  ## Whether to report per-cpu stats or not
  percpu = true
  ## Whether to report total system cpu stats or not
```

```
  totalcpu = true
  ## If true, collect raw CPU time metrics.
  collect_cpu_time = false
  ## If true, compute and report the sum of all non-idle CPU states.
  report_active = false


# Read metrics about disk usage by mount point
[[inputs.disk]]
  ## By default, telegraf gather stats for all mountpoints.
  ## Setting mountpoints will restrict the stats to the specified mountpoints.
  # mount_points = ["/"]

  ## Ignore some mountpoints by filesystem type. For example (dev)tmpfs (usually
  ## present on /run, /var/run, /dev/shm or /dev).
  ignore_fs = ["tmpfs", "devtmpfs", "devfs"]


# Read metrics about disk IO by device
[[inputs.diskio]]
  ## By default, telegraf will gather stats for all devices including
  ## disk partitions.
  ## Setting devices will restrict the stats to the specified devices.
  # devices = ["sda", "sdb"]
  ## Uncomment the following line if you need disk serial numbers.
  # skip_serial_number = false
  #
  ## On systems which support it, device metadata can be added in the form of
  ## tags.
  ## Currently only Linux is supported via udev properties. You can view
  ## available properties for a device by running:
  ## 'udevadm info -q property -n /dev/sda'
  # device_tags = ["ID_FS_TYPE", "ID_FS_USAGE"]
  #
  ## Using the same metadata source as device_tags, you can also customize the
  ## name of the device via templates.
  ## The 'name_templates' parameter is a list of templates to try and apply to
  ## the device. The template may contain variables in the form of '$PROPERTY' or
  ## '${PROPERTY}'. The first template which does not contain any variables not
  ## present for the device is used as the device name tag.
  ## The typical use case is for LVM volumes, to get the VG/LV name instead of
  ## the near-meaningless DM-0 name.
  # name_templates = ["$ID_FS_LABEL","$DM_VG_NAME/$DM_LV_NAME"]


# Get kernel statistics from /proc/stat
[[inputs.kernel]]
  # 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 19: 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 20: 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 21: 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}}' +
           ↪  messageN + '{{end}}')
34         .id(idVar)
35         .idTag(idTag)
36         .levelTag(levelTag)
37         .messageField(messageField)
38         .durationField(durationField)
39         .slack()
40         .channel('#embassyalerts')
41 {% endraw %}
42         .exec('butler_healing_agent', 'relaunch-worker', '-t', '{{
           ↪  pillar['terraform_files'] }}', '-s', '{{ pillar['terraform_state']
           ↪  }}', '-v', '{{ pillar['terraform_vars'] }}', '-p', '{{
           ↪  pillar['terraform_provider'] }}')
43 {% raw %}
44 trigger
45     |eval(lambda: "emitted")
46         .as('value')
47         .keep('value', messageField, durationField)
48     |influxDBOut()
49         .create()
50         .database(outputDB)
51         .retentionPolicy(outputRP)
52         .measurement(outputMeasurement)
53         .tag('alertName', name)
54         .tag('triggerType', triggerType)
55
56 trigger
57     |httpOut('output')
```

```
{% endraw %}
```

Listing 22: Butler healing agent code for restarting the Airflow Scheduler.

```
def call_command(command, cwd=None):
  try:
      logging.debug("About to invoke command: " + command)
      my_output = check_output(command, shell=True, cwd=cwd, stderr=STDOUT)
      logging.debug("Command output is: " + my_output)
      return my_output
  except CalledProcessError as e:
      logging.error("An error occurred! Command output is: " +
      ↪  e.output.decode("utf-8") )
      raise

def is_critical(level):
  return level == "CRITICAL"

def parse_alert_data():
  return json.loads(sys.stdin.read())

def get_host_name(alert_data):
  return alert_data["data"]["series"][0]["tags"]["host"]

def restart_service(host, service_name):
  call_command("pepper {} service.restart {}".format(host, service_name), None)

def parse_args():
  my_parser = argparse.ArgumentParser()

  sub_parsers = my_parser.add_subparsers()

  common_args_parser = argparse.ArgumentParser(
      add_help=False, conflict_handler='resolve')

  restart_airflow_scheduler_parser = sub_parsers.add_parser(
      "restart-airflow-scheduler", parents=[common_args_parser],
      ↪  conflict_handler='resolve')

  ↪  restart_airflow_scheduler_parser.set_defaults(func=restart_airflow_scheduler_command)

def restart_airflow_scheduler_command(args, alert_data):
  if is_critical(alert_data["level"]):
      restart_service("-G 'roles:tracker'", "airflow-scheduler")
```

Listing 23: Butler healing agent code for relaunching a failed VM.

```
def call_command(command, cwd=None):
  try:
      logging.debug("About to invoke command: " + command)
      my_output = check_output(command, shell=True, cwd=cwd, stderr=STDOUT)
      logging.debug("Command output is: " + my_output)
      return my_output
  except CalledProcessError as e:
      logging.error("An error occurred! Command output is: " +
      ↪  e.output.decode("utf-8") )
      raise

def is_critical(level):
  return level == "CRITICAL"

def parse_alert_data():
  return json.loads(sys.stdin.read())

def get_host_name(alert_data):
  return alert_data["data"]["series"][0]["tags"]["host"]

def restart_service(host, service_name):
  call_command("pepper {} service.restart {}".format(host, service_name), None)

def parse_args():
  my_parser = argparse.ArgumentParser()

  sub_parsers = my_parser.add_subparsers()

  common_args_parser = argparse.ArgumentParser(
    add_help=False, conflict_handler='resolve')

  relaunch_worker_parser = sub_parsers.add_parser(
      "relaunch-worker", parents=[common_args_parser],
      ↪  conflict_handler='resolve')
  relaunch_worker_parser.add_argument(
      "-t", "--terraform_location", help="Location of the terraform definition
      ↪  files.",
      dest="terraform_location", required=True)
  relaunch_worker_parser.add_argument(
      "-s", "--terraform_state_location", help="Location of the terraform state
      ↪  file.",
      dest="terraform_state_location", required=True)
  relaunch_worker_parser.add_argument(
      "-v", "--terraform_var_file_location", help="Location of the terraform vars
      ↪  file.",
      dest="terraform_var_file_location", required=True)
  relaunch_worker_parser.add_argument(
      "-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_minon_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
            ↪  master.".format(minion_connect_try, MINION_CONNECT_MAX_RETRIES,
            ↪  host_name))
64          key_data = call_command("pepper --client=wheel key.name_match
            ↪  match={}".format(host_name))
65          logging.debug("Retrieved key data: " + key_data)
66          if is_key_present(key_data, host_name):
67              return True
68          else:
69              logging.debug("Key data for host {} not found at time {}. Sleeping for
                ↪  {} seconds.".format(host_name, datetime.now(),
                ↪  MINION_CONNECT_SLEEP_PERIOD))
70              time.sleep(MINION_CONNECT_SLEEP_PERIOD)
71              minion_connect_try = minion_connect_try + 1
72
73      return False
74
75  def relaunch_worker_command(args, alert_data):
76      if is_critical(alert_data["level"]):
77          host_name = get_host_name(alert_data)
78
79
80          tf_location = args.terraform_location
81          tf_state_location = args.terraform_state_location
82          tf_var_file_location = args.terraform_var_file_location
83          tf_resource = provider_resource_lookup[args.terraform_provider]
84          worker_number = host_name.split("-")[1]
85
86          call_command("pepper --client=wheel key.delete match={}".format(host_name))
```

```
87        call_command("terraform taint -lock=false -state={}
          ↪  {}.worker.{}".format(tf_state_location, tf_resource, worker_number),
          ↪  tf_location)
88        call_command("terraform apply -lock=false -state={} --var-file {}
          ↪  -auto-approve".format(tf_state_location, tf_var_file_location),
          ↪  tf_location)
89
90        locate_minon_key(host_name)
91
92        call_command("pepper '*' mine.update")
93        call_command("pepper {} state.apply dnsmasq".format(host_name))
94        call_command("pepper {} state.apply consul".format(host_name))
95        call_command("pepper {} state.highstate".format(host_name))
```

Listing 24: 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
                   ↪  --report-genotype-likelihood-max"
7          }
8  }
```

Listing 25: 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,
                    ↪  creating.".format(result_path_prefix))
35              os.makedirs(result_path_prefix)
36
37          result_filename = "{}/{}_{}.vcf".format(
38              result_path_prefix, sample_id, contig_name)
39
40          freebayes_path = config["freebayes"]["path"]
41          freebayes_mode = config["freebayes"]["mode"]
42          freebayes_flags = config["freebayes"]["flags"]
43
44          reference_location = config["reference_location"]
45
46          if freebayes_flags == None:
47              freebayes_flags = ""
48
49          if freebayes_mode == "discovery":
50              freebayes_command = "{} -r {} -f {} {} {} > {}".\
51                  format(freebayes_path,
52                         contig_name,
53                         reference_location,
54                         freebayes_flags,
55                         sample_location,
56                         result_filename)
57          elif freebayes_mode == "regenotyping":
58              variants_location = config["variants_location"]
59
60              freebayes_command = "{} -r {} -f {} -@ {} {} {} > {}".\
61                  format(freebayes_path,
62                         contig_name,
63                         reference_location,
64                         variants_location[contig_name],
```

```
                        freebayes_flags,
                        sample_location,
                        result_filename)
        else:
             raise ValueError("Unknown or missing freebayes_mode -
             ↪  {}".format(freebayes_mode))

        call_command(freebayes_command, "freebayes")

        compressed_sample_filename = compress_sample(result_filename, config)
        generate_tabix(compressed_sample_filename, config)
        copy_result(compressed_sample_filename, sample_id, config)
    else:
        logger.info(
            "Contig {} is not in the contig whitelist,
            ↪  skipping.".format(contig_name))


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime.datetime(2020, 01, 01),
    'email': ['airflow@airflow.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG("freebayes", default_args=default_args,
          schedule_interval=None, concurrency=10000, max_active_runs=2000)


start_analysis_run_task = PythonOperator(
    task_id="start_analysis_run",
    python_callable=start_analysis_run,
    provide_context=True,
    dag=dag)


validate_sample_task = PythonOperator(
    task_id="validate_sample",
    python_callable=validate_sample,
    provide_context=True,
    dag=dag)

validate_sample_task.set_upstream(start_analysis_run_task)

```

```
complete_analysis_run_task = PythonOperator(
    task_id="complete_analysis_run",
    python_callable=complete_analysis_run,
    provide_context=True,
    dag=dag)

for contig_name in tracker.util.workflow_common.CONTIG_NAMES:
    freebayes_task = PythonOperator(
        task_id="freebayes_" + contig_name,
        python_callable=run_freebayes,
        op_kwargs={"contig_name": contig_name},
        provide_context=True,
        dag=dag)

    freebayes_task.set_upstream(validate_sample_task)

    complete_analysis_run_task.set_upstream(freebayes_task)
```

Listing 26: Saltstack state for workflow deployment.

```
current_runs = run_session.query(Configuration.config[("sample","
↪  sample_id")].astext).\
        join(AnalysisRun, AnalysisRun.config_id == Configuration.config_id).\
        join(Analysis, Analysis.analysis_id == AnalysisRun.analysis_id).\
        filter(and_(Analysis.analysis_id == analysis_id, AnalysisRun.run_status
        ↪  != tracker.model.analysis_run.RUN_STATUS_ERROR)).all()

available_samples = sample_session.query(PCAWGSample.index.label("index"),
↪  sample_id.label("sample_id"), sample_location.label("sample_location")).\
        join(SampleLocation, PCAWGSample.index == SampleLocation.donor_index).\
        filter(and_(sample_location != None, sample_id.notin_(current_runs))).\
        limit(num_runs).all()
```

Listing 27: Butler Analysis configuration for SNP genotyping.

```
{
        "variants_location": {
            "1": "/freebayes.chr_1.sites.snv_indel.annot.final.vcf.gz",
            "2": "/freebayes.chr_2.sites.snv_indel.annot.final.vcf.gz",
            "3": "/freebayes.chr_3.sites.snv_indel.annot.final.vcf.gz",
            "4": "/freebayes.chr_4.sites.snv_indel.annot.final.vcf.gz",
            "5": "/freebayes.chr_5.sites.snv_indel.annot.final.vcf.gz",
            "6": "/freebayes.chr_6.sites.snv_indel.annot.final.vcf.gz",
            "7": "/freebayes.chr_7.sites.snv_indel.annot.final.vcf.gz",
            "8": "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
```

```
            "9": "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
            "10": "/freebayes.chr_10.sites.snv_indel.annot.final.vcf.gz",
            "11": "/freebayes.chr_11.sites.snv_indel.annot.final.vcf.gz",
            "12": "/freebayes.chr_12.sites.snv_indel.annot.final.vcf.gz",
            "13": "/freebayes.chr_13.sites.snv_indel.annot.final.vcf.gz",
            "14": "/freebayes.chr_14.sites.snv_indel.annot.final.vcf.gz",
            "15": "/freebayes.chr_15.sites.snv_indel.annot.final.vcf.gz",
            "16": "/freebayes.chr_16.sites.snv_indel.annot.final.vcf.gz",
            "17": "/freebayes.chr_17.sites.snv_indel.annot.final.vcf.gz",
            "18": "/freebayes.chr_18.sites.snv_indel.annot.final.vcf.gz",
            "19": "/freebayes.chr_19.sites.snv_indel.annot.final.vcf.gz",
            "20": "/freebayes.chr_20.sites.snv_indel.annot.final.vcf.gz",
            "21": "/freebayes.chr_21.sites.snv_indel.annot.final.vcf.gz",
            "22": "/freebayes.chr_22.sites.snv_indel.annot.final.vcf.gz",
            "X": "/freebayes.chr_X.sites.snv_indel.annot.final.vcf.gz",
            "Y": "/freebayes.chr_Y.sites.snv_indel.annot.final.vcf.gz"
        },
        "results_base_path":
        ↪  "/shared/data/results/regenotype_freebayes_discovery/",
        "results_local_path": "/tmp/regenotype_freebayes_discovery/",
        "freebayes": {
                "mode": "regenotyping",
                "flags": "-l"
        }
}
```

Listing 28: Butler Workflow configuration for Data Submission.

```
{
        "gnos": {
                "ebi": {
                        "url": "https://gtrepo-ebi.annailabs.com"
                },
                "osdc_icgc": {
                        "url": "https://gtrepo-osdc-icgc.annailabs.com"
                },
                "osdc_tcga": {
                        "url": "https://gtrepo-osdc-tcga.annailabs.com"
                }
        },
        "rsync": {
                "flags": "-a -v --remove-source-files"
        }
}
```

Listing 29: Butler Analysis configuration for Data Submission.

```
{
        "gnos": {
                "ebi": {
                        "key_location":
                        ↪  "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.pem"
                },
                "osdc_icgc": {
                        "key_location":
                        ↪  "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.pem"
                },
                "osdc_tcga": {
                        "key_location":
                        ↪  "/home/airflow/.ssh/sergei_bionimbus_gnos_may.pem"
                }
        },
        "metadata_template_location": "/opt/pcawg-germline/workflows/gtupload-wo
        ↪  rkflow/analysis_template.xml",
        "submission_base_path":
        ↪  "/shared/data/results/freebayes_discovery_gnos_submission/",
        "destination_repo_mapping": {
                "ICGC": "ebi",
                "TCGA": "osdc_tcga"
        }
}
```

Listing 30: Python code for the run_delly function which implements the functionality of the delly_genotype task inside the Butler Delly Workflow.

```
def run_delly(**kwargs):

    config = get_config(kwargs)
    sample = get_sample(kwargs)

    sample_id = sample["sample_id"]
    sample_location = sample["sample_location"]

    result_path_prefix = config["results_local_path"] + "/" + sample_id

    if (not os.path.isdir(result_path_prefix)):
        logger.info(
            "Results directory {} not present,
            ↪  creating.".format(result_path_prefix))
        os.makedirs(result_path_prefix)

    delly_path = config["delly"]["path"]
    reference_location = config["reference_location"]
```

```
    variants_location = config["variants_location"]
    variants_type = config["delly"]["exclude_template_path"]

    result_filename = "{}/{}_{}.bcf".format(
        result_path_prefix, sample_id, variants_type)

    log_filename = "{}/{}_{}.log".format(
        result_path_prefix, sample_id, variants_type)

    delly_command = "{} call -t {} -g {} -v {} -o {} -x {} {} > {}".\
        format(delly_path,
               variants_type,
               reference_location,
               variants_location,
               result_filename,
               exclude_template_path,
               sample_location,
               log_filename)

    call_command(delly_command, "delly")

    copy_result(result_filename, sample_id, config)
```

Listing 31: Butler Delly Workflow analysis configuration to genotype deletions.

```
{
        "variants_location": "/delly_deletion_sites/del.sites.bcf",
        "results_base_path":
        ↪  "/shared/data/results/delly_germline_deletions_14_07_2016/",
        "results_local_path": "/tmp/delly_germline_deletions/",
        "variants_type": "DEL"

}
```

Bibliography

[1] URL: https://www.influxdata.com/time-series-platform/telegraf/.

[2] URL: https://github.com/etsy/statsd.

[3] URL: https://github.com/influxdata/kapacitor.

[4] URL: https://slack.com.

[5] URL: https://github.com/influxdata/chronograf.

[6] Enis Afgan et al. "Galaxy CloudMan: delivering cloud compute clusters". In: *BMC bioinformatics* 11.12 (2010), p. 1.

[7] Charu C Aggarwal. *Data streams: models and algorithms*. Vol. 31. Springer Science & Business Media, 2007.

[8] 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.

[9] Can Alkan, Bradley P Coe, and Evan E Eichler. "Genome structural variation discovery and genotyping". In: *Nature reviews. Genetics* 12.5 (2011), p. 363.

[10] 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.

[11] *Apache Airflow (incubating) Documentation — Airflow Documentation*. URL: https://airflow.incubator.apache.org/ (visited on 10/31/2016).

[12] 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.

[13] Shay Banon. *Elasticsearch*. URL: https://www.elastic.co/products/elasticsearch (visited on 10/13/2016).

[14] Ralph M Barnes. "Motion and time study." In: (1949).

[15] James O Berger. *Statistical decision theory and Bayesian analysis*. Springer Science & Business Media, 2013.

[16] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[17] 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.

[18] Benoit G Bruneau. "The developmental genetics of congenital heart disease". In: *Nature* 451.7181 (2008), pp. 943–948.

[19] 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.

[20] Kristian Cibulskis et al. "ContEst: estimating cross-contamination of human samples in next-generation sequencing data". In: *Bioinformatics* 27.18 (2011), pp. 2601–2602.

[21] 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.

[22] 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.

[23] Francis S Collins and Harold Varmus. "A new initiative on precision medicine". In: *New England Journal of Medicine* 372.9 (2015), pp. 793–795.

[24] 1000 Genomes Project Consortium et al. "A map of human genome variation from population-scale sequencing". In: *Nature* 467.7319 (2010), pp. 1061–1073.

[25] 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.

[26] ENCODE Project Consortium et al. "An integrated encyclopedia of DNA elements in the human genome". In: *Nature* 489.7414 (2012), pp. 57–74.

[27] 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.

[28] Georgiana Copil et al. "Multi-level elasticity control of cloud services". In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 429–436.

[29] 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.

[30] Armon Dadgar. *Consul*. HashiCorp. URL: `https://www.consul.io/` (visited on 10/13/2016).

[31] Petr Danecek et al. "The variant call format and VCFtools". In: *Bioinformatics* 27.15 (2011), pp. 2156–2158.

[32] Charles Darwin. *On the origin of species*. D. Appleton and Co., 1871. DOI: `10.5962/bhl.title.28875`.

[33] Mayur Datar et al. "Maintaining stream statistics over sliding windows". In: *SIAM journal on computing* 31.6 (2002), pp. 1794–1813.

[34] J Davis II et al. *Overview of the Ptolemy project*. Tech. rep. ERL Technical Report UCB/ERL, 1999.

[35] 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.

[36] 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.

[37] Wil MP van Der Aalst et al. "Workflow patterns". In: *Distributed and parallel databases* 14.1 (2003), pp. 5–51.

[38] Juliane 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.

[39] Richard Durbin et al. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[40] Genomics England. "The 100,000 genomes project". In: *The* 100 (2016), pp. 1–2.

[41] Thomas Erl. *Service-oriented architecture (SOA): concepts, technology, and design*. 2005.

[42] Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action*. Manning Greenwich, 2011.

[43] *European Open Science Cloud | Open Science - Research and Innovation - European Commission*. URL: `http://ec.europa.eu/research/openscience/index.cfm?pg=open-science-cloud` (visited on 10/31/2016).

[44] 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.

[45] 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.

[46] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[47] 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.

[48] Rosalind E Franklin and Raymond G Gosling. "Molecular configuration in sodium thymonucleate". In: *Nature* 171 (1953), pp. 740–741.

[49] 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.

[50] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. "Mining data streams: a review". In: *ACM Sigmod Record* 34.2 (2005), pp. 18–26.

[51] 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.

[52] Erik Garrison and Gabor Marth. "Haplotype-based variant detection from short-read sequencing". In: *arXiv preprint arXiv:1207.3907* (2012).

[53] *Glossary V1 - EGIWiki*. URL: `https://wiki.egi.eu/wiki/Glossary_V1` (visited on 10/28/2016).

[54] 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.

[55] Christopher Greenman et al. "Patterns of somatic mutation in human cancer genomes". In: *Nature* 446.7132 (2007), pp. 153–158.

[56] Mendel Gregor. "Versuche uber Pflanzen-Hybriden. Verhandlungen des naturforschenden Vereines in Brunn". In: 4 (1865), pp. 3–47.

[57] 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.

[58] Daniel F Gudbjartsson et al. "Large-scale whole-genome sequencing of the Icelandic population". In: *Nature genetics* 47.5 (2015), pp. 435–444.

[59] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[60] Douglas Hanahan and Robert A Weinberg. "Hallmarks of cancer: the next generation". In: *cell* 144.5 (2011), pp. 646–674.

[61] Robert L Henderson. "Job scheduling under the portable batch system". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1995, pp. 279–294.

[62] David Hollingsworth. "The workflow reference model". In: (1995).

[63] *Home*. URL: `https://www.denbi.de/` (visited on 10/31/2016).

[64] *Homepage | Celery: Distributed Task Queue*. URL: `http://www.celeryproject.org/` (visited on 10/31/2016).

[65] 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.

[66] *InfluxDB – Time-Series Data Storage*. InfluxData. URL: `https://www.influxdata.com/time-series-platform/influxdb/` (visited on 10/12/2016).

[67] Mark Jobling, Matthew Hurles, and Chris Tyler-Smith. *Human evolutionary genetics: origins, peoples & disease*. Garland Science, 2013.

[68] Peter A Jones and Stephen B Baylin. "The epigenomics of cancer". In: *Cell* 128.4 (2007), pp. 683–692.

[69] Jocelyn Kaiser and Jennifer Couzin-Frankel. "Biden seeks clear course for his cancer moonshot". In: *Science* 351.6271 (2016), pp. 325–326.

[70] *Kibana*. URL: `https://www.elastic.co/products/kibana` (visited on 10/13/2016).

[71] Bartha Maria Knoppers and Ruth Chadwick. “Human genetic research: emerging trends in ethics”. In: *Nature Reviews Genetics* 6.1 (2005), pp. 75–79.

[72] Eric S Lander et al. “Initial sequencing and analysis of the human genome”. In: *Nature* 409.6822 (2001), pp. 860–921.

[73] Ben Langmead and Steven L Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature methods* 9.4 (2012), pp. 357–359.

[74] Michael S Lawrence et al. “Discovery and saturation analysis of cancer genes across 21 tumour types”. In: *Nature* 505.7484 (2014), pp. 495–501.

[75] Hane Lee et al. “Clinical exome sequencing for genetic identification of rare Mendelian disorders”. In: *Jama* 312.18 (2014), pp. 1880–1887.

[76] Monkol Lek et al. “Analysis of protein-coding genetic variation in 60,706 humans”. In: *Nature* 536.7616 (2016), pp. 285–291.

[77] Reuven M Lerner. “At the forge: PostgreSQL, the NoSQL database”. In: *Linux Journal* 2014.247 (2014), p. 5.

[78] Heng Li. “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”. In: *arXiv preprint arXiv:1303.3997* (2013).

[79] Heng Li. “Towards better understanding of artifacts in variant calling from high-coverage samples”. In: *Bioinformatics* (2014), btu356.

[80] Heng Li and Richard Durbin. “Fast and accurate short read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760.

[81] 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.

[82] Heng Li et al. “The sequence alignment/map format and SAMtools”. In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.

[83] Bertram Ludäscher et al. “Scientific workflow management and the Kepler system”. In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.

[84] 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.

[85] 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.

[86] 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.

[87] Elaine R Mardis. “Next-generation DNA sequencing methods”. In: *Annu. Rev. Genomics Hum. Genet.* 9 (2008), pp. 387–402.

[88] 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.

[89] Vivien Marx. “Biology: The big challenges of big data”. In: *Nature* 498.7453 (2013), pp. 255–260.

[90] Vivien Marx. “The DNA of a nation”. In: *Nature* 524.7566 (2015), pp. 503–505.

[91] 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.

[92] William McLaren et al. “The ensembl variant effect predictor”. In: *Genome biology* 17.1 (2016), p. 122.

[93] Peter Mell and Tim Grance. “The NIST definition of cloud computing”. In: (2011).

[94] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.

[95] 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.

[96] 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.

[97] Todd K Moon. “The expectation-maximization algorithm”. In: *IEEE Signal processing magazine* 13.6 (1996), pp. 47–60.

[98] 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.

[99] Shanmugavelayutham Muthukrishnan et al. “Data streams: Algorithms and applications”. In: *Foundations and Trends® in Theoretical Computer Science* 1.2 (2005), pp. 117–236.

[100] Michael W Nachman. “Single nucleotide polymorphisms and recombination rate in humans”. In: *TRENDS in Genetics* 17.9 (2001), pp. 481–485.

[101] Rasmus Nielsen et al. “Genotype and SNP calling from next-generation sequencing data”. In: *Nature Reviews Genetics* 12.6 (2011), pp. 443–451.

[102] 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.

[103] Patrick O’Neil et al. “The log-structured merge-tree (LSM-tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[104] Torkel Ödegaard. *Grafana.net*. URL: `https://grafana.net/` (visited on 10/12/2016).

[105] Tom Oinn et al. “Taverna: a tool for the composition and enactment of bioinformatics workflows”. In: *Bioinformatics* 20.17 (2004), pp. 3045–3054.

[106] 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.

[107] Stavros Papadopoulos et al. “The TileDB array data storage manager”. In: *Proceedings of the VLDB Endowment* 10.4 (2016), pp. 349–360.

[108] 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.

[109] 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.

[110] 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.

[111] James L Peterson. “Petri net theory and the modeling of systems”. In: (1981).

[112] Erin D Pleasance et al. “A comprehensive catalogue of somatic mutations from a human cancer genome”. In: *Nature* 463.7278 (2010), p. 191.

[113] *RabbitMQ - Messaging that just works*. URL: `https://www.rabbitmq.com/` (visited on 10/31/2016).

[114] Tobias Rausch et al. “DELLY: structural variant discovery by integrated paired-end and split-read analysis”. In: *Bioinformatics* 28.18 (2012), pp. i333–i339.

[115] 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.

[116] Nicola D Roberts et al. “A comparative analysis of algorithms for somatic SNV detection in cancer”. In: *Bioinformatics* (2013), btt375.

[117] Dan Robinson et al. “Integrative clinical genomics of advanced prostate cancer”. In: *Cell* 161.5 (2015), pp. 1215–1228.

[118] Richard M Russell. “The CRAY-1 computer system”. In: *Communications of the ACM* 21.1 (1978), pp. 63–72.

[119] *SaltStack automation for CloudOps, ITOps, and DevOps at scale*. URL: `https://saltstack.com/` (visited on 10/31/2016).

[120] 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.

[121] 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.

[122] Nitin Sawant and Himanshu Shah. “Big Data Access Patterns”. In: *Big Data Application Architecture Q & A*. Springer, 2013, pp. 57–68.

[123] Stephan C Schuster. “Next-generation sequencing transforms today’s biology”. In: *Nature* 200.8 (2007), pp. 16–18.

[124] 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).

[125] Dennis J Selkoe. "Amyloid β-protein and the genetics of Alzheimer's disease". In: *Journal of Biological Chemistry* 271.31 (1996), pp. 18295–18298.

[126] Robert Shapiro. "A technical comparison of XPDL, BPML and BPEL4WS". In: *Cape Visions* (2002).

[127] Jordan Sissel. *Logstash*. URL: `https://www.elastic.co/products/logstash` (visited on 10/13/2016).

[128] *Start page – collectd – The system statistics collection daemon*. URL: `https://collectd.org/` (visited on 10/31/2016).

[129] Lincoln D Stein et al. "Data analysis: create a cloud commons". In: *Nature* 523 (2015), pp. 149–151.

[130] Zachary D Stephens et al. "Big data: astronomical or genomical". In: *PLoS biology* 13.7 (2015), e1002195.

[131] Michael R Stratton, Peter J Campbell, and P Andrew Futreal. "The cancer genome". In: *Nature* 458.7239 (2009), pp. 719–724.

[132] Peter H Sudmant et al. "An integrated map of structural variation in 2,504 human genomes". In: *Nature* 526.7571 (2015), pp. 75–81.

[133] Adrian Tan, Gonçalo R Abecasis, and Hyun Min Kang. "Unified representation of genetic variants". In: *Bioinformatics* (2015), btv112.

[134] Frederick Winslow Taylor. *Scientific management*. Routledge, 2004.

[135] *Terraform by HashiCorp*. URL: `https://www.terraform.io/` (visited on 10/31/2016).

[136] *The Cost of Sequencing a Human Genome - National Human Genome Research Institute (NHGRI)*. URL: `https://www.genome.gov/sequencingcosts/` (visited on 11/04/2016).

[137] Lindsey A Torre et al. "Global cancer statistics, 2012". In: *CA: a cancer journal for clinicians* 65.2 (2015), pp. 87–108.

[138] 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.

[139] J Craig Venter et al. "The sequence of the human genome". In: *science* 291.5507 (2001), pp. 1304–1351.

[140] J Craig Venter et al. "Shotgun sequencing of the human genome". In: *Science* 280.5369 (1998), pp. 1540–1542.

[141] Kathleen A Vermeersch and Mark P Styczynski. "Applications of metabolomics in cancer research". In: *Journal of carcinogenesis* 12 (2013).

[142] 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.

[143] 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.

[144] Zhong Wang, Mark Gerstein, and Michael Snyder. "RNA-Seq: a revolutionary tool for transcriptomics". In: *Nature reviews genetics* 10.1 (2009), pp. 57–63.

[145] James D Watson, Francis HC Crick, et al. "Molecular structure of nucleic acids". In: *Nature* 171.4356 (1953), pp. 737–738.

[146] 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.

[147] John N Weinstein et al. "The cancer genome atlas pan-cancer analysis project". In: *Nature genetics* 45.10 (2013), pp. 1113–1120.

[148] 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.

[149] Justin P Whalley et al. "Framework For Quality Assessment Of Whole Genome, Cancer Sequences". In: *bioRxiv* (2017), p. 140921.

[150] Christopher Wilks et al. "The Cancer Genomics Hub (CGHub): overcoming cancer through the power of torrential data". In: *Database* 2014 (2014), bau093.

[151] 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.

[152] Christina K Yung et al. "ICGC in the cloud". In: *Cancer Research* 76.14 Supplement (2016), pp. 3605–3605.

[153] Shelemyahu Zacks. *The theory of statistical inference*. Vol. 34. Wiley New York, 1971.

[154] 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.

[155] 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.

[156] Songnian Zhou. "Lsf: Load sharing in large heterogeneous distributed systems". In: *I Workshop on Cluster Computing*. Vol. 136. 1992.