THE UNIVERSITY OF CHICAGO

HYBRID-CLOUD SCHEDULING FOR LONG-RUNNING BIOINFORMATICS WORKFLOWS UNDER TIME CONSTRAINTS

A THESIS SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
MARTIN PUTRA

CHICAGO, ILLINOIS
DECEMBER 2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This thesis is done under the supervision of Drs. Robert L. Grossman and Haryadi S. Gunawi of the University of Chicago and in collaboration with In Kee Kim of the University of Georgia. This thesis work was supported by NCI-GDC.

ABSTRACT

The amount of genomics data is increasing at a rapid pace. This leads to a situation where high-throughput data processing is needed to efficiently mine insights from the increasingly abundant genomics data. However, bioinformatics workflows/jobs typically take large input files (e.g., hundreds of GB), resulting in jobs having long execution times.

This thesis work seeks to answer the following research question: "How to ensure X% deadline satisfaction when scheduling bioinformatics workflows in hybrid clouds?". To address the problem, this thesis presents a novel hybrid cloud scheduler that significantly reduces deadline miss rate at no or minimal extra cost. The new scheduler consists of two main components: 1) an accurate execution time predictor for bioinformatics workflows and 2) a novel scheduling policy for ensuring a high statistical guarantee of completing job executions on spot instances leveraging spot lifetime distributions and job duration prediction.

In this thesis, I developed a high-fidelity simulator that accurately resembles the behaviors of bioinformatics workflows on hybrid cloud environments composed of on-premise and cloud machines. The proposed scheduler was modeled on top of the high-fidelity simulator and evaluated using large-scale, real-world production traces from one of the leading genomics research centers. The evaluation results show that the new scheduler successfully reduced deadline miss rates and/or cost compared to several baseline scheduling policies. Finally, the thesis outlined future research directions for refining and implementing the scheduler on production-level genomics processing systems.

CHAPTER 1

INTRODUCTION

The amount of genomics data is increasing at a rapid pace [1, 2]. By 2025, it is projected to be more than, or at least on par with, the traditionally conceived biggest contributors of 'big data' such as Astronomy, Twitter, and Youtube [3]. The growth is often attributed to the continuous invention of many sequencing techniques and platforms, starting with Sanger's DNA Sequencing technique in the late 1970s [4] up to the recent single molecule sequencing platform [5], with each becoming subsequently faster and cheaper. This leads to a situation where high-throughput data processing is needed to efficiently mine insights from the *increasingly* abundant genomics data.

Genomics data processing is commonly done as a chain of special-purpose tools [6]. For example, a common DNA sequence alignment process will include FastQC [7] for filtering/trimming raw input, BWA [8] for alignment to genome reference, and Picard [9] for post-alignment analysis. This chain of tools forms a directed acyclic graph (DAG), where each node in the DAG represents one tool, and the output of one node becomes the input for another tool [10]. Such a chain of tools is also known as *workflow*. In practice, bioinformatics workflows are usually written using domain-specific languages, such as Nextflow [11] and Common Workflow Language (CWL) [12]. Many genomics research centers open-sourced the workflows they have written [13, 14].

Bioinformatics workflows/jobs[1] typically take large input files (e.g., hundreds of GB), which becomes the main contributor to having a long execution time. We consider the problem of executing such long-running bioinformatics jobs given deadline. In particular, we seek to answer the following research question: *"How to ensure X% deadline satisfaction when scheduling bioinformatics workflows in hybrid cloud?"* A straightforward approach to satisfying a job/workflow's deadline is to use additional resources in the cloud infrastructure

1. In this thesis, term *workflow* and *job* are used interchangeably.

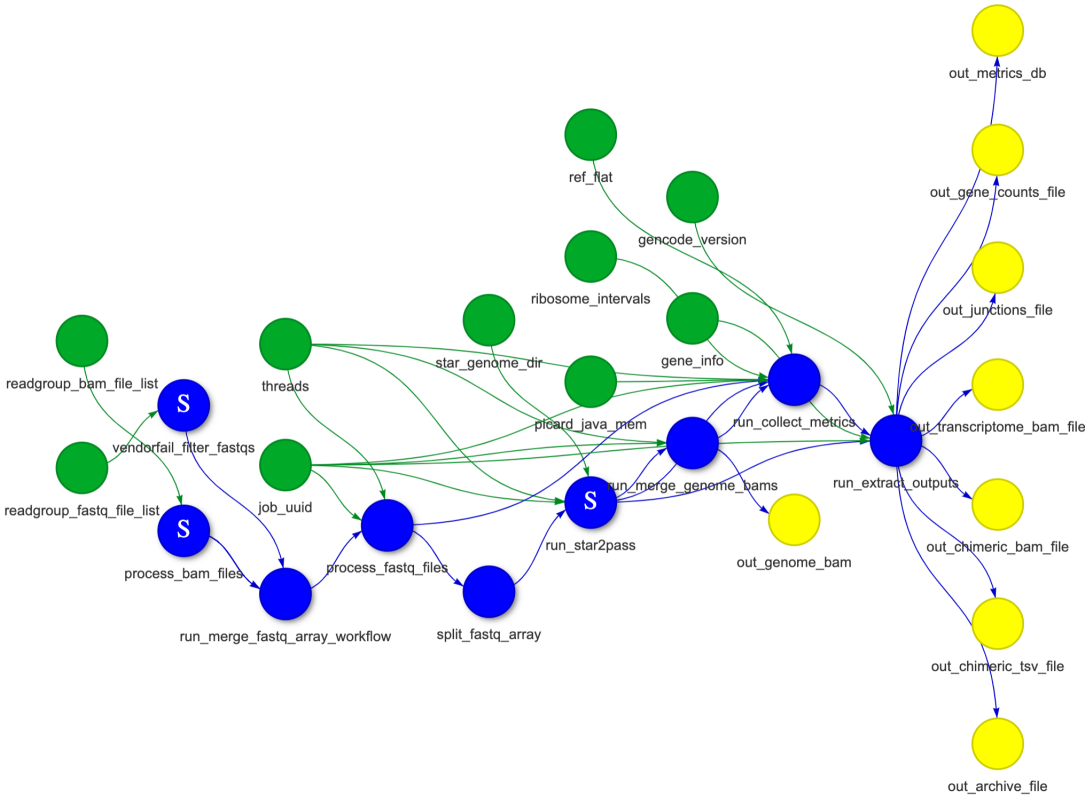


Figure 1.1: An example of production workflow used by NCI-GDC (National Cancer Institute-Genomic Data Commons), one of the leading genomics research centers. Green nodes represent input, yellow nodes represent output, and blue nodes represent one or more bioinformatics tools.

by leveraging the cloud's elasticity and scalability [15, 16]. However, adding extra resources comes at costs; therefore we also need to develop the approach in a cost-efficient manner.

To address this problem, this thesis develops novel hybrid cloud scheduling for long-running bioinformatics workflows with deadlines. There are multiple research challenges when designing this new scheduling algorithm. First, job execution time prediction is considered essential for a performant scheduler [17], yet accurately estimating job execution time remains difficult [18, 19]. Accurately estimating job execution time is particularly challenging for bioinformatics workflows because such workflows often show long-tail behaviors. Second, the scheduling algorithm should ensure cost efficiency. In particular, our scheduler attempts to leverage transient VM[2] models (e.g., AWS spot), which are significantly cheaper than typical on-demand virtual machines (VMs) in clouds [20], However, spot VMs do not offer the same level of availability as on-demand VMs (e.g., 99.99% of uptime [21]), and cloud providers can revoke the spot VMs anytime with short notice [22, 23, 24].

To address the first challenge, this thesis empirically shows the essential features required to build a high-accuracy job execution time predictor. We use those features to train two popular models, namely Random Forest [25] and XGBoost [26], then show that it is possible to get over 95% prediction accuracy for long-running bioinformatics jobs. To address the second challenge, this thesis research develops an approach to reliably executing long-running jobs on spot instances. Specifically, we leverage the job execution time predictor and historical data for spot instances' lifetime to proactively guide the scheduler to place jobs on spot instances, which have a high statistical guarantee of completing the given bioinformatics jobs.

We evaluate the new scheduler using a high-fidelity, discrete event hybrid cloud simulator written in Python. The simulation used a 6 months-long production cluster trace from one of the leading genomics research centers. Moreover, the simulator includes two important

2. In this thesis, the terms "transient," "preemptible," and "spot" instances are used *interchangeably*. Additionally, the term "instance" and "virtual machine (VM)" are also used *interchangeably*.

components: a.) our novel machine learning predictor and b.) a realistic failure model based on real-world datasets from the same production cluster. The simulator allows us to validate our scheduler design against real-life scenarios in a quick and affordable manner. Our evaluation results show that our algorithm significantly reduces deadline miss rate with no or very little extra cost.

This paper makes the following contributions:

- We develop a novel scheduling algorithm for long-running bioinformatics workflows on a hybrid cloud under time constraints.
- We develop an accurate job execution time predictor for long-running bioinformatics workflows.
- We show how to leverage our job execution time predictor to build a cost-efficient scheduler, which minimizes deadline, even when cloud VMs (e.g., spot VMs) can be revoked anytime.
- We developed a high-fidelity hybrid cloud simulator for bioinformatics workflow scheduling with large-scale and real-world production traces.

The remainder of this paper is organized as follows. chapter 2 presents background on bioinformatics workflows and describes related works on workflow scheduling. This thesis presents the new scheduling algorithm, along with the design of our simulator, on chapter 3 and chapter 4. This thesis, then, discusses the evaluation results on chapter 5. Finally, chapter 6 concludes this thesis and outlines possible future directions.

CHAPTER 2

BACKGROUND AND RELATED WORKS

This chapter presents background and prior works related to bioinformatics workflow schedulings, especially in the hybrid cloud setup. We first briefly discuss genomics processing at a higher level, then describe *GDC Pipeline Automation System* (GPAS), the genomics processing system we study to build our new scheduler. GPAS and its historical workload will serve as the environment in which this thesis models its new scheduler and simulator. Next, the chapter will briefly discuss the characteristics of bioinformatics job executions observed in the GPAS dataset and traces. Finally, the chapter discusses relevant prior works.

2.1 Background

2.1.1 Genome Processing

At a high level, genomics data processing can be classified into three categories: primary, secondary, and tertiary analysis [5]. During primary analysis, genomic samples are processed using special-purpose hardware called *sequencing platform* [27], which determines the category of each base nucleotide within the sample (i.e., **T** for thymine, **A** for adenine, **C** for cytosine, and **G** for guanine). The result is what is usually called a *read file*. The read file mainly consists of T, C, G, and A letters, along with a score describing how confident the hardware is for each particular letter. Then, quality control software [7] will trim the low confidence bases. This *trimmed read file* (hereafter will be referred to simply as *read file*) becomes the input for subsequent secondary and tertiary analyses.

During secondary analysis, a set of bioinformatics applications, e.g., SAMtools [28], GATK [29], are used to align read files to reference human genomes and identify differences between them. This process is known as *variant calling* [30]. *Variant calling* is crucial in order to locate possible mutations (i.e. *variants*) within the genome sample. Finally, dur-

ing the tertiary analysis, another set of bioinformatics tools, such as Variant Effect Predictor (VEP) [31] and ANNOVAR [32], are used to infer how those variants express themselves on a patient's physical condition.

2.1.2 GDC Pipeline Automation System (GPAS)

The NCI-GDC (National Cancer Institute-Genomic Data Commons)[1], commonly referred to as GDC, is one of the leading genomics research centers. GDC operates a *genomics data commons* [33], a platform that enables the cancer research community to store, analyze, and share genomics data. In terms of analyzing data, GDC receives requests from users to run certain workflows on a set of input files. All GDC workflows are open-sourced [13] and publicly available on GitHub[2].

To fulfill users' demands, GDC maintains an in-house genomics workflow processing system called *GPAS*. GPAS was built with hybrid cluster in mind. It supports computation and storage either locally on onprem machines or remotely in the cloud. As an automated genomics processing system, GPAS automatically pulls various input files and relevant genomics reference files from local and/or remote storage, executes the requested workflows, then stores the results locally or remotely (Figure 2.1). To date GDC has processed more than 3 petabytes of data from more than 80.000 cases across 60 NCI-funded projects [34].

GDC operates an on-premise data center to provide the computational resources necessary for meeting users' demands. The data center consists of several clusters. The production cluster is composed of 80 bare metal and 140 VM machines, each having around 200 GBs memory, 40 cores @ 2.5GHz, and 3 TBs storage with various mixtures of HDD and SSD. The production cluster uses SLURM [35, 36] as its resource manager. The data center also provides development and testing clusters. However, it might not be necessary to go into

1. `https://gdc.cancer.gov/`

2. `https://github.com/NCI-GDC/gdc-workflow-overview.git`

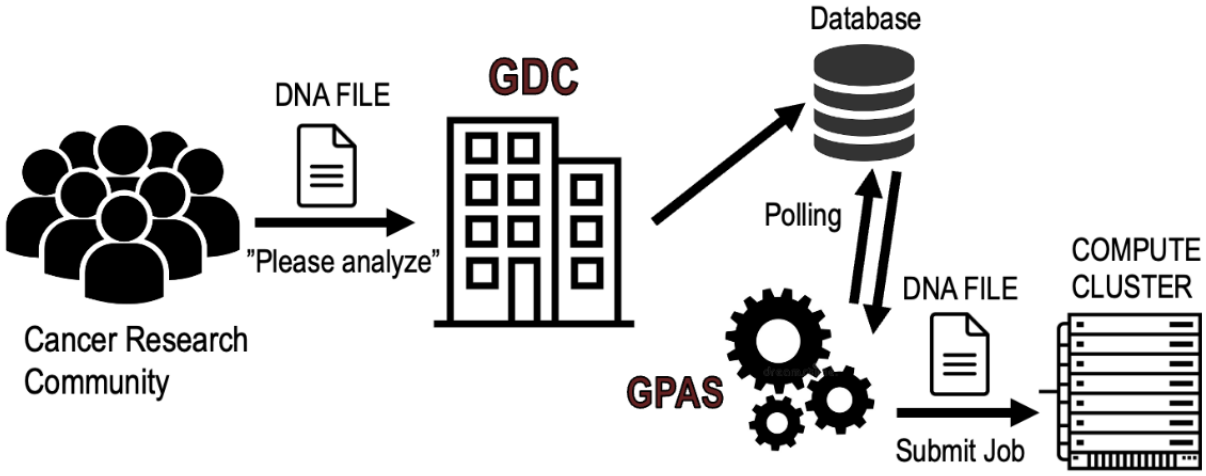


Figure 2.1: GDC receives analysis requests from user (cancer research community) and submit them as bioinformatics jobs to GPAS.

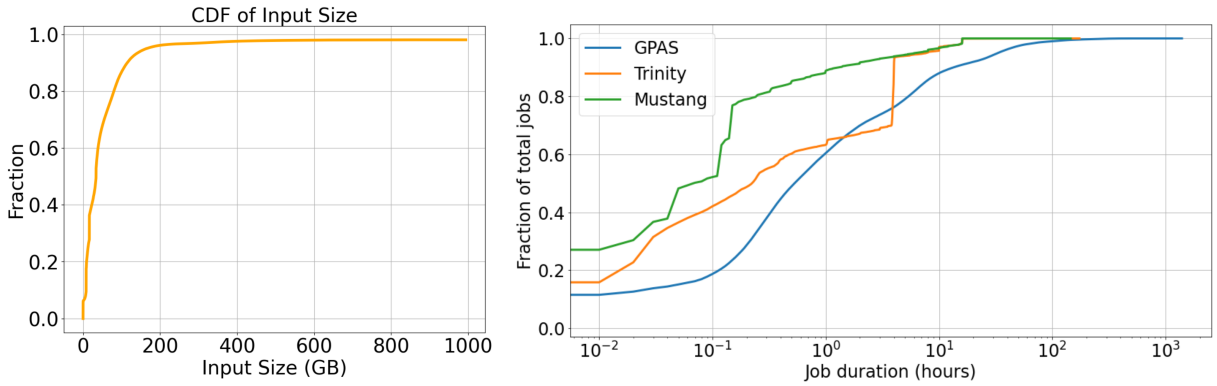
their details as this thesis will only use datasets from GDC's production cluster.

Despite the massive scale of cloud data center resources, there are times when the incoming demand is projected to exceed the capacity of available resources for a brief period of time. This phenomenon is historically known as flash crowd [37], or more recently, workload spikes [38, 39]. Systems operators had a long-leveraged idea of "spare resources which can be utilized on-demand" to address such precarious situations [40]. With the advent of cloud computing, this technique found its natural complement owing to the cloud's illusion of infinite resources and elasticity. Thus the technique became an established pattern in managing compute clusters and is now known as *cloudbursting* [41, 42] leveraging both on-premise computing resources and on-demand cloud VMs.

GPAS employs this cloudbursting technique to handle occasional workload spikes. It does so by utilizing SLURM's cloudbursting capability [43], which supports the provisioning/deprovisioning cloud VMs (in AWS, MS Azure, and Google Cloud) and job executions on those cloud VMs with similar operational interfaces as onpremise job submissions. System operators need to create a SLURM configuration defining at least two clusters, one for onpremise and another for cloud, and then connect the cloud cluster to the chosen cloud

Job Execution Time (Unit: Hour)	Bioinformatics Jobs	High-Perf. Comp. Jobs	Capability Comp. Jobs
Mean	99.6	8.6	7.3
Standard Dev.	89.7	7.5	13.9
25%ile	36.0	4.0	2.1
50%ile	73.3	8.1	4.8
75%ile	132.6	12.2	9.1
Max	1395.3	145.3	173.6

Table 2.1: Statistical description of job execution time for bioinformatics (GPAS), HPC (Mustang), and capability computing (Trinity) jobs. Capability computing refers to the type of computation which uses novel hardware technologies to solve the most demanding problems [44].



(a) Input data size

(b) Job execution time

Figure 2.2: Characteristics of a sample bioinformatics workload. On (a), we see that it is common for bioinformatics jobs to have hundreds of GBs input size. Consequently, (b) shows how bioinformatics job's execution time (GPAS) often exceeded those of HPC (Mustang [44]) and capability computing jobs.

provider. The cloud cluster initially has zero VMs when no jobs are submitted. Then, once a job is submitted to the cloud cluster, SLURM will provision a pre-configured type of VM and execute the job on that cloud VM. Additionally, the cloud VM will be automatically terminated when no more jobs are in SLURM's queue.

2.1.3 Bioinformatics Jobs Characteristics

Due to its large input data size and complex data analysis pipelines, bioinformatics jobs often have long execution times (e.g., tens to hundreds of hours). Figure 2.2 compares the execution time of HPC, capability computing, and bioinformatics jobs. The HPC and capability computing jobs use workload traces from Mustang and Trinity systems[44], two compute clusters in Los Alamos National Laboratory (LANL)[3]. Mustang and Trinity's traces were released in 2018. Meanwhile, the bioinformatics jobs are represented by GPAS production cluster traces collected for more than two years between 2018-2021. It is observed that, while the 95^{th} percentile of HPC jobs finish in < 10 hours, that time only accounts for the 88^{th} percentile of bioinformatics jobs. From a reliability perspective, this characteristic presents a challenge for system operators. Such long-running jobs are much more prone to error and/or failure[4].

The jobs also suffer from very long tail. As shown in Figure 2.2, 1.5% of bioinformatics jobs have tail ranging up to 19.1× of median job execution time, which translates to almost two months of turnaround. This is in contrast to HPC (Mustang) and capability computing (Trinity) jobs, where all but a few jobs[5] finish before 25 and 31 hours, respectively. Table 2.1 shows the statistical summary among the three types of workload.

3. `https://www.lanl.gov/`

4. The recovery capability of bioinformatics jobs depends significantly on the type of software used throughout the execution, which is not available on GPAS. While one might argue that this problem can easily be addressed by migrating to software that provides good recovery capability, anecdotal experiences shared by domain experts show that, in reality, it is not trivial to make such changes. Thus, although we believe this direction is worth pursuing, we consider it outside the scope of this thesis. We assume jobs do not have means for checkpointing nor post-failure recovery other than restarting from scratch.

5. Our analysis on the released trace shows that only 3 (out of 1620) Mustang jobs have >100 hours execution time, while Trinity only has 8 (out of 843).

2.1.4 Hybrid Cloud and Spot Instances

Hybrid cloud refers to a system that, by default, executes jobs on onprem cluster (or private cloud), yet utilizes cloud elasticity and autoscaling during workload spikes in order to successfully handle suddenly increased demands [45, 41]. A hybrid cloud system faces several challenges when offloading tasks/jobs to the public cloud due to the variety of resources/VM offerings in the clouds. In public clouds, e.g., AWS, MS Azure, and Google Cloud, it is common to offer various/specialized VMs for user workloads, including compute-optimized, storage-optimized, and memory-optimized VMs. This opens up a possibility for the system to rent machines based on the dominant computation resources required by its workload. The cloud also offers various offerings based on the raw amount of resources offered. The compute, storage, and memory-optimized categories are further combined with size categories such as *small, medium*, and *large* [46]. This multitude of options presents considerable challenges for system operators/researchers in choosing the right type of offerings for their particular workload and system configurations. On the other hand, these options clearly present opportunities for those who are able to leverage it, and thus the community has extensively studied this area [47, 48, 49, 50].

A variety of options exist not only in terms of performance/types of VM but also in terms of lease types. This choice space affects how much control users have over the availability of their rented VMs. Commonly, there are three types of VM lease offerings: ondemand, reserved, and preemptible instances (e.g., AWS Spot VMs). Ondemand instance can be instantiated anytime and kept alive as long as required. Users pay based on the duration of time the machine is alive. Thus, ondemand instance provides the best flexibility and availability, yet it is usually the most expensive among the three [51]. Reserved instances are rented on long-term contracts (e.g., one month or year). Users usually pay upfront, and the instances will be kept alive throughout the agreed period. Compared to ondemand instances, reserved instances are usually cheaper when both are kept alive for the same amount of time,

yet they necessitate the user to sacrifice some flexibility. Finally, preemptible instances are the cheapest among the three options. This type of instance can also be instantiated anytime, but the cloud provider retains the right to revoke them anytime with very short notice (typically 2 minutes for AWS spot), whether there are ongoing computations or not. It is the users' responsibility to ensure that all data and computation have been saved by the time of revocation. This model presents both an opportunity and challenge for system operators, especially for those intending to run long-running jobs on the cloud, since it provides the biggest saving potential yet the least availability. This thesis mainly considers using these preemptible instances to run long-running bioinformatics jobs.

2.2 Related Work

2.2.1 Workflow Scheduling

As stated in Section 2.1.3, GPAS does not support checkpointing or post-recovery failure mechanisms, and it restarts the jobs from scratch when facing job execution failure. This is mainly due to the limitation of existing bioinformatics workflow execution software, which is incapable of breaking down the execution of workflow DAG into per-node execution. Currently, GPAS executes the whole DAG as one job on one machine. This limitation makes many relevant workflow scheduling works unapplicable for our case.

Nevertheless, the problem is still an active area of research among the systems community. There are at least two factors fostering this sustained interest. First, the DAG computation model is still prevalent in the large-scale production cluster of leading industries. Several papers describing the inner workings of large-scale production systems showed that it presents both challenges and opportunities from a systems perspective. For example, Microsoft [52] and Alibaba [53, 54] have disclosed that their workloads have a DAG-like structure. The recurrent nature of tasks in the DAG provides the necessary information for task execution

time estimation, which is immensely useful for performant scheduling [55, 56, 57, 19]. On the other hand, the heterogeneity of the tasks means a good scheduler has to consider different types of resources to execute the tasks optimally [58, 59, 60, 61, 62, 63]. Second, there has been renewed interest in collecting, analyzing, and sharing large-scale workload traces [64, 65, 66, 67, 68, 69, 53, 44, 54], and many of them reveal the DAG computation model. These traces become more popular, so will the DAG computation model, and more work will be done to understand and take advantage of its property.

Several works have been done on scheduling DAG jobs in the cloud [70, 71, 42, 23]. Stratus [42] attempts to reduce workflow execution cost in the cloud by ensuring that VMs are either alive with as close to 100% utilization as possible or letting them be 0% utilized instead so they can be immediately terminated. To ensure maximum utilization, Stratus leverages task execution time predictor such that tasks with similar end times (task completion times) are executed on the same machine. Unfortunately, this scheduling mechanism often leads to 0% utilization by the time those tasks finish. If it does not, Stratus either move tasks to other instances whose ongoing tasks finish around the same time or instantiates a new instance as a last resort. However, Stratus assumes the ability to migrate task execution from one instance to another, which does not exist in our case. Hourglass [23] strives to minimize cloud execution cost while satisfying deadlines by using a mix of ondemand and spot resources. The key insight is to identify the latest time a job has to be sent to an ondemand instance before violating the deadline, then exploit the slack time for making as much computation progress as possible using the cheap spot instances. When the job runs out of slack time, the scheduler will send the job to ondemand instance. Similar to Stratus, Hourglass also assumes migration in addition to checkpointing capability.

Other works emphasized the importance of runtime estimation for workflow scheduling. 3Sigma [19] represents each job in a Space-Time Request Language (SRTL) which tells the scheduler both the space constraints (e.g. resource type, placement constraints) and time

constraints (e.g. deadlines, latency SLO) in an algebraic language. The scheduler then uses the full historical distribution of each task's execution, rather than single-point estimates (e.g., mean, median, 95%ile of execution time), to provide accurate estimation and guide the scheduler against misprediction. The main insight is task distribution provides more information about overprediction and underprediction compared to single-point estimates. SLearn [18] claims that history-based prediction techniques can lead to suboptimal scheduling decisions because the assumption that history-based prediction – *jobs can be recurrent and have similar properties* – may not always be true. A potential alternative is to *learn over space* by leveraging DAG jobs' spatial dimension. SLearn uses the execution time of several pilot tasks to provide estimates of other tasks in the DAG while carefully handling delays and variations due to task heterogeneity.

2.2.2 Current GPAS scheduling

GPAS employs different scheduling mechanisms between the onpremise resources and the cloud VM cluster. The onpremise cluster uses SLURM's default FIFO scheduling, while the cloud cluster requires system operators' intervention to manually decide when to execute jobs and which instances should be used. Thus, it is important that system operators should be able to estimate how long a job will take and what kind of resources (e.g., memory, CPU, storage) it needs the most. System operators are also required to ensure cloudbursting cost does not exceed a certain monthly budget. These decisions are made solely based on the systems operator's analysis and judgment over historical data.

CHAPTER 3

PROPOSED SCHEDULER DESIGN

In essence, a scheduling decision consists of two aspects, namely *time* and *space*. The *time* aspect includes whether to immediately start or delay the execution of a job and if so, for how long. The reasons for delaying execution might range from one which is not avoidable, such as unavailable resources, to one which is deliberate, such as optimistically waiting for a 'better' resource [72]. The *space* aspect includes on which resource a job should be executed. This decision spans several factors, such as locality, processing rate, and cost per time unit. In our context, this decision includes whether to execute jobs on onprem, ondemand, or spot machines.

This chapter describes a new scheduler designed to execute long-running bioinformatics jobs in the cloud. The proposed scheduler attempts to address two goals, namely: 1) minimizing deadline miss rate and 2) enforcing cost efficiency. To minimize the deadline miss rate of bioinformatics jobs, the scheduler leverages accurate prediction to estimate the expected job execution time on computing resources or waiting time if a job is put in a queue. This information is used to decide whether the scheduler should cut the job's waiting time by offloading it to the cloud. Furthermore, to achieve cost efficiency, the scheduler uses preemptible spot instances rather than ondemand whenever possible. This policy presents the scheduler with the challenge of choosing which jobs are suitable for execution on spot instances. The scheduler addresses this challenge by considering the spot lifetime distribution; then, it executes a job that provides a high statistical guarantee of completion within the expected spot instance's lifetime.

The chapter proceeds as follows. This chapter first describes the execution time predictor models, namely the Random Forest [25] and XGBoost models [26], and compares their prediction accuracy. Following that, this chapter describes how to choose *spot threshold*, a threshold on execution time which is essential for choosing which jobs can be reliably

Feature Name	Feature Type		r	Description
	Numerical	Categorical		
alloc_gres_scratch	✓	-	0.67	Amount of storage allocated by SLURM
alloc_tres_cpu	✓	-	0.14	Amount of cores allocated by SLURM
alloc_tres_mem	✓	-	0.20	Amount of memory allocated by SLURM
alloc_tres_node	✓	-	-	Number of cluster nodes allocated by SLURM
arr_time_sec	✓	-	-0.07	Job arrival time in GPAS
file_size_gb	✓	-	0.42	Total size of input files (including genomics references)
workflow_uuid	-	✓	-	Identifier of GDC workflow used for the job

Table 3.1: Handpicked features for building execution time predictor, along with their types, description, and Pearson's correlation r. Correlation is calculated on every numerical features except *alloc_tres_node*, whose values are identical and cause the formula to divide by zero.

executed on spot VMs. Finally, this chapter concludes with how a scheduler can use the two components to minimize deadline miss rate while enforcing cost efficiency.

3.1 Execution Time Predictor

To provide an accurate estimation of job execution time, I implemented two decision-tree based regression models, namely Random Forest [25] and XGBoost [26] using `scikit-learn` machine learning library [73]. The model implementation process started with identifying the most important features in the dataset.

3.1.1 Feature Selection

It is not possible to describe the dataset I use in detail due to confidentiality reasons. The dataset consists of information regarding requests made by users to GDC, the genomics files associated with those requests, and the SLURM metrics for jobs executing those requests. I manually determined a set of features based on my communication with GDC's bioinformaticians and system operators, then employed two techniques to confirm the handpicked features' potential to affect job execution time. Table 3.1 shows the description and types of each feature.

For clarity purposes, I will use capital letters to denote a vector and lower-case letters

for a single data point. Let $Y = [y_1, y_2, ..., y_p]^T$ be a $1 \times p$ vector of job execution time, where y_i denotes a single execution time for $1 \leq i \leq p$. Let also $N = \{N_1, N_2, ..., N_m\}$ be the set of all numerical features in the dataset, where similar to job execution time, $N_i = [n_{i1}, n_{i2}, ..., n_{ip}]^T$ is a $1 \times p$ vector.

The two techniques are as follows. First, for each numerical feature $V \in N$, I calculated Pearson's correlation coefficient r between feature V and job execution time Y. Pearson's Correlation Coefficient [74] between two $1 \times p$ vector V and Y is defined as :

$$r = \frac{\sum_i (v_i - \bar{v})(y_i - \bar{y})}{\sqrt{\sum_i (v_i - \bar{v})^2 \sum_i (y_i - \bar{y})^2}}, \quad 1 \leq i \leq p \tag{3.1}$$

where v_i, y_i are the i-th element of vector V and Y, and $\bar{v}, \bar{y}$ are the mean of vector V and Y, respectively. Then, using the value of r for each pair of feature V and execution time Y, it is possible to build a correlation coefficient heatmap to visualize their relationships. Figure 3.1 shows the resulting heatmap. It can be inferred that features related to file input size yield the highest correlation to job execution time.

Second, for the categorical variable *worfklow_ uuid*, I visualized the quantiles of execution time for each value to inspect their similarity. The resulting box plot is shown in Figure 3.2. It can be seen that *workflow_ uuid* can be a good feature candidate, as indicated by the variety of execution times across the sampled workflows.

3.1.2 Hyperparameter Tuning and Performance Comparison

The next steps to build the model that includes coding and hyperparameter tuning. Since I used an off-the-shelf machine learning library for both Random Forest and XGBoost models, the coding process is trivial and will not be described further. For hyperparameter tuning, Random Forest and XGBoost are based on Decision Tree, thus it is not surprising that both models have several identical hyperparameters. Due to interest of time, I only evaluated

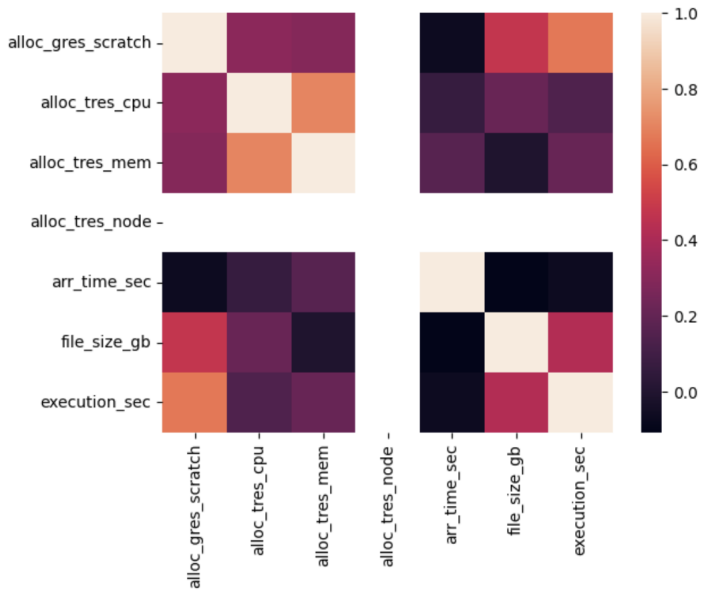


Figure 3.1: Heatmap of Pearson's correlation coefficient among the numerical features. The job execution time is represented by *execution_ sec*.

two parameters, namely the number of trees (*Num. Estimators*) and maximum depth of the trees (*Max. Depth*) within each model.

I first performed a preliminary investigation of the possibility of adverse effects on performance from each hyperparameter. This is done by training each model with an extremely high parameter value to find the 'limit' on which the parameter either stops increasing the model's performance or even starts to decrease it. From this process, I found that a good value range for *Num. Estimators* is $1 - 25$, while for *Max. Depth* it is $1 - 50$. Following that, I conducted an exhaustive search over all possible combinations of *(Num. Estimators, Max. Depth)* value pair to find one that yields the best performance. The metrics used to evaluate each configuration are *Coefficient of Determination* (R^2), *Root Mean Squared Error (RMSE)*, and *Mean Absolute Percentage Error (MAPE)*.

Coefficient of Determination is a well-accepted metric to determine the fitness of a regression/prediction model. It represents the variance of samples adequately described by the variance of model predictions. Given a $1 \times p$ vector of observed values Y, its mean $\bar{y}$, and

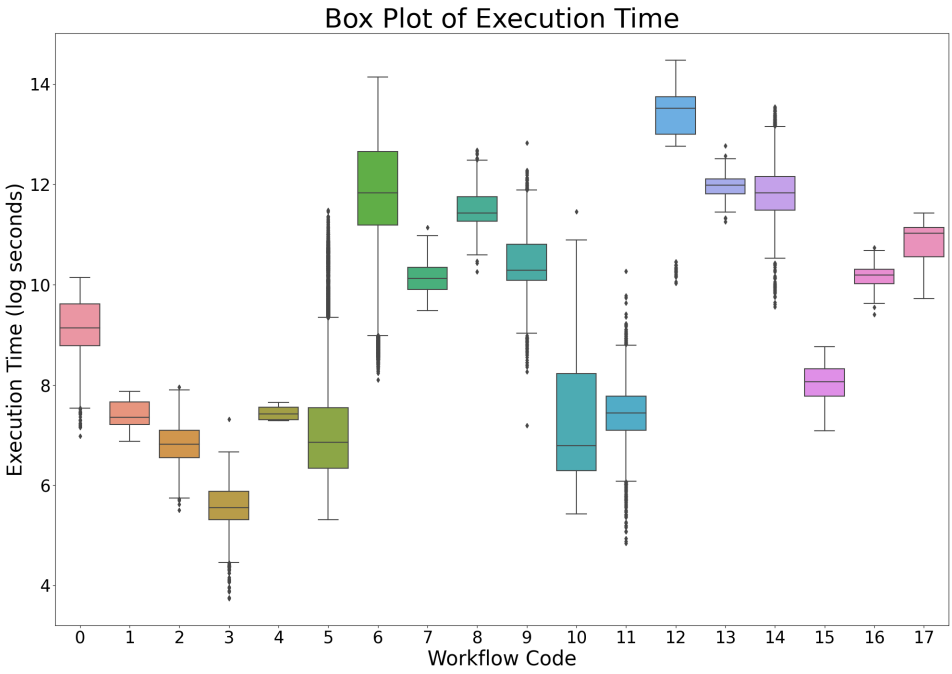


Figure 3.2: Box plot of execution time for 20 workflows, sampled randomly out of 101 workflows in the dataset. For each box, the lines indicate 0^{th} (i.e. min), 25^{th}, 50^{th} (i.e. median), 75^{th}, and the 100^{th} (i.e. max) percentile for that workflow. Each UUID is encoded with an ordinal category due to interest of space. As can be seen in the Figure, the sampled workflows have vastly different percentiles, which might imply workflow type is a good feature candidate for the predictor.

the respective $1 \times p$ vector of predicted values $\hat{Y}$, *Coefficient of Determination* is defined as:

$$R^2 = \frac{\textit{Explained Variation}}{\textit{Total Variation}} = \frac{\sum_i (\hat{y_i} - \bar{y})^2}{\sum_i (y_i - \bar{y})^2}, \quad 1 \leq i \leq p \tag{3.2}$$

Root Mean Squared Error (RMSE) measures how far predicted values fall from observed values and is defined as:

$$RMSE = \sqrt{\frac{\sum_i (\hat{y_i} - y_i)^2}{p}} \tag{3.3}$$

While *Mean Absolute Percentage Error (MAPE)* measures the mispredictions relative to the observed values:

$$MAPE = \frac{1}{p} \sum_i \left| \frac{\hat{y_i} - y_i}{y_i} \right|, \quad 1 \leq i \leq p \tag{3.4}$$

Table 3.2 shows the resulting metrics for a selection of *Max. Depth* and *Num. Estimators* values. Random Forest can achieve better performance than XGBoost using smaller *Max. Depth* and *Num. Estimators* values, for example, $(10, 3)$, which usually translates to faster training time. However, once given adequate values, such as $(25, 50)$, XGBoost exceeds the performance of Random Forest. It is interesting to note that Random Forest's performance seems to be more affected by *Max. Depth* rather than *Num. Estimators*, but the opposite is true for XGBoost.

3.2 Failure Model

It is important to accurately model spot revocation behavior to reliably design the scheduler on real-world situations. I am aware of at least three approaches to achieving this goal. The first and ideal way would be to base our model on spot instances' revocation data from public clouds (e.g., AWS). The challenge here lies in finding good quality dataset with an adequate

Hyperparameter (Max Depth, Num. Estimators)	Random Forest			XGBoost		
	R^2(%)	RMSE	MAPE	R^2(%)	RMSE	MAPE
(2,3)	53.78	63586.40	11.88	48.32	67386.59	9.29
(2,5)	54.16	63525.91	11.84	58.83	60253.43	9.53
(2,10)	54.29	63350.34	11.98	69.15	52407.92	9.59
(2,50)	54.23	63366.10	11.98	81.18	40268.43	4.04
(10,3)	90.01	29215.46	1.92	78.32	43966.71	1.23
(10,5)	90.48	28555.21	1.92	88.28	32539.30	1.31
(10,10)	90.61	28501.59	2.05	92.11	26808.07	0.93
(10,50)	90.88	28313.60	2.06	92.59	26482.76	0.66
(25,3)	90.61	28197.71	0.38	79.15	41983.38	0.38
(25,5)	91.24	27085.76	0.375	88.76	30712.34	0.32
(25,10)	91.74	25884.82	0.44	91.77	26550.77	0.33
(25,50)	91.4	27192.28	0.34	92.69	25680.78	0.45

Table 3.2: Testing scores of Random Forest and XGBoost models using various combination of max depth and number estimators. Reported metrics include *accuracy*, *root mean square error (RMSE)*, and *mean absolute percentage error (MAPE)*. Overall, XGBoost shows better performance compared to Random Forest, with best result shown by max depth = 25 and number of estimators = 50.

amount of sample points. The second way is to assume the existence of a bidding market for spot instances, such as done in [75]. A user bids a certain price for each spot instance, and the instance is revoked whenever the current market price exceeds the user's bid price. Based on this assumption, a spot lifetime distribution is created using the historical AWS Spot Market dataset. The last way is to assume a certain distribution (e.g., normal) and statistical properties (e.g., mean and standard deviation). This approach can be useful for quick prototyping, but its assumptions must be rigorously proven in order to provide reliable evaluation results.

This thesis leveraged the second approach to generate a spot lifetime distribution using an open-source AWS Spot Market dataset [76]. The dataset records spot pricing changes for various availability zones, regions, and instance types. For resembling the revocation behavior, I bid 1.3× of the market price at any point in time, then record the duration before the market price exceeds the bid. The resulting lifetime distribution is shown in Figure 3.3 together with the distribution of GPAS job durations.

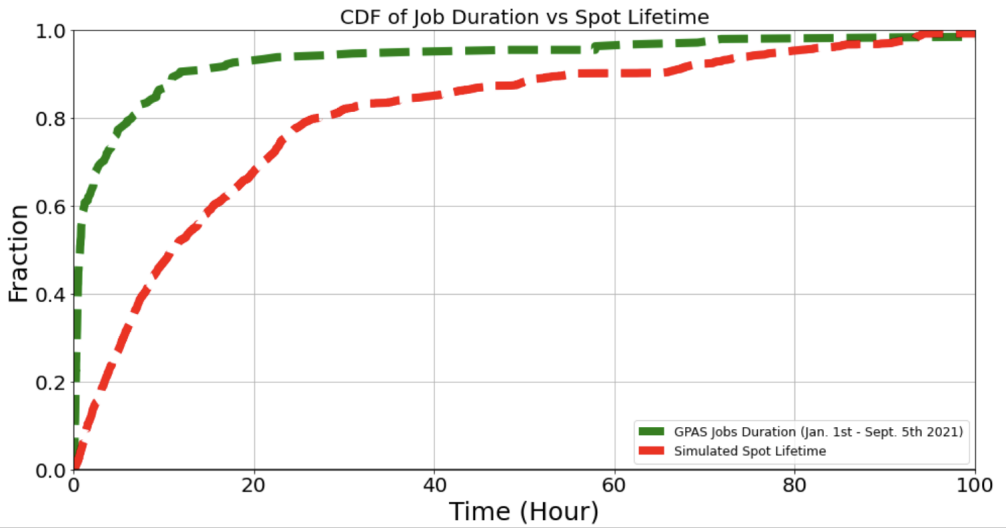


Figure 3.3: CDF of GPAS jobs duration shown together with simulated lifetime. The lifetime is generated by artificially bidding 1.3× of market price at any point in time using an open-sourced AWS Spot Market dataset [76].

3.3 Threshold for Sending Jobs to Preemptible Spot Cloud VMs

Figure 3.3 shows a noticeable difference between spot instances' lifetimes and GPAS job execution times. For example, while around 90% of GPAS jobs have a duration of up to 10 hours, only 50% of spot instances can live up to that period (10 hours). Given this observation, this thesis asks the following question: how can a scheduler use this information to decide whether it should execute an incoming job on a spot instance? In particular, can we find a threshold (decision point) t such that if jobs with a duration less than t are sent to a spot instance, the jobs will have a high probability of finishing successfully? This approach will allow the scheduler to cloudburst jobs cost-efficiently while minimizing the effect of spot instance failures.

To determine such threshold, let us use a hypothetical situation where N new jobs arrived at the system. Each job will be considered for execution on an onprem, ondemand, or spot instance. Let d denote job duration. If the scheduler decides to execute a job on the spot instance, it will instantiate a spot instance with lifetime l. Let F_D and F_L denote the CDF of GPAS job duration and spot instance's lifetime as shown in Figure 3.3. It is also

assumed that d and l are randomly sampled from the distribution represented by F_D and F_L, respectively.

First, I describe the expected number of jobs sent to spot instances by the scheduler. Given CDF of jobs duration F_D, N incoming jobs, and a certain threshold t, the expected number of jobs sent to spot E_{spot} is:

$$E_{spot}(t) = F_D(t) \cdot N \tag{3.5}$$

These jobs might fail depending on whether their duration is longer than the instantiated spot lifetime. In order to express the expected number of successful jobs, let D be a random variable of job duration, while L is a random variable of spot lifetime. The expression needs to capture two cases. First, a job will finish successfully if the spot lifetime is longer than the threshold (since job duration is always less than the threshold). Second, when both spot lifetime and job duration are shorter than the threshold, the probability of the job's successful completion is $P(L > D | L < t, D < t)$. Thus, we can express the expectation of successful jobs as:

$$E_{success} = E_{spot}(t) \cdot (1 - F_L(t)) + E_{spot}(t) \cdot P(L > D | L < t, D < t) \tag{3.6}$$

$$= E_{spot}(t) \cdot (1 - F_L(t)) + E_{spot}(t) \cdot \int^{d=t} F_D(t) f_L(t) \quad \mathbf{d}d \tag{3.7}$$

The first term, which includes $(1-F_L(t))$, takes into account the instantiated spot lifetime being bigger than threshold t. The second term considers $L < t, D < t$, where it is difficult to ensure whether the spot lifetime will be longer than the job duration.

The explanation above is necessary because it corresponds to the intuition for choosing threshold value t. As expressed in Equation 3.7, the expected number of successful jobs increases by maximizing $F_D(t)$ (within E_{spot}) and minimizing $F_L(t)$ (within $(1 - F_L(t))$. Simply choosing t for one term without considering the other does not work. For example,

choosing t solely for minimizing $F_L(t)$ will also decrease $F_D(t)$, resulting in fewer jobs sent to spot instances. This leads to jobs being sent either to ondemand (increasing cloud costs) or to onprem (increasing job waiting time in the queue).

One intuitive way to choose t is to consider the 'tail turning point' among the two CDFs, such as $t = 10$ for the CDF of GPAS job duration. Taking $t = 10$ allows 90^{th} of jobs to be send to spot while giving us $(1 - F_L(t)) = 0.5$. This provides a reasonable trade-off between maximizing $F_D(t)$ and minimizing $F_L(t)$. Unless noted otherwise, the remainder of this thesis uses $t = 10$ as the threshold for sending jobs to spot instances.

3.4 JSDQ on Hybrid Cloud

Building upon the machine learning predictor and failure model, I designed JSDQ, a hybrid cloud scheduling algorithm, which minimizes deadline miss rate while ensuring cost-efficiency. The main insights underlying JSDQ are as follows; 1) leverage spot instances whenever possible and 2) carefully choose jobs, which are sent to spot such that they have a high probability of finishing. The first strategy allows the algorithm to save cost due to spot instances' low prices compared to the price of ondemand instances. The second strategy minimizes the risk of deadline misses and cost increases due to job failures.

Figure 3.4 shows the workflow of JSDQ. On job arrival, the scheduler will first predict job execution time using one of the implemented models. Unless otherwise noted, the remainder of this thesis will use XGBoost as it provides better prediction. Given job execution time prediction, JSDQ uses the historical dataset to classify jobs into three categories, namely small, medium, and large jobs, which are defined as the 75^{th}, 95^{th}, and 98^{th} percentile, respectively. JSDQ then sends small and medium jobs to cloud, while large jobs are considered for onprem machines. Readers might notice that the thresholds for small and medium jobs follow the analysis in Section 3.3, which outlines how to choose threshold t for sending jobs to spot.

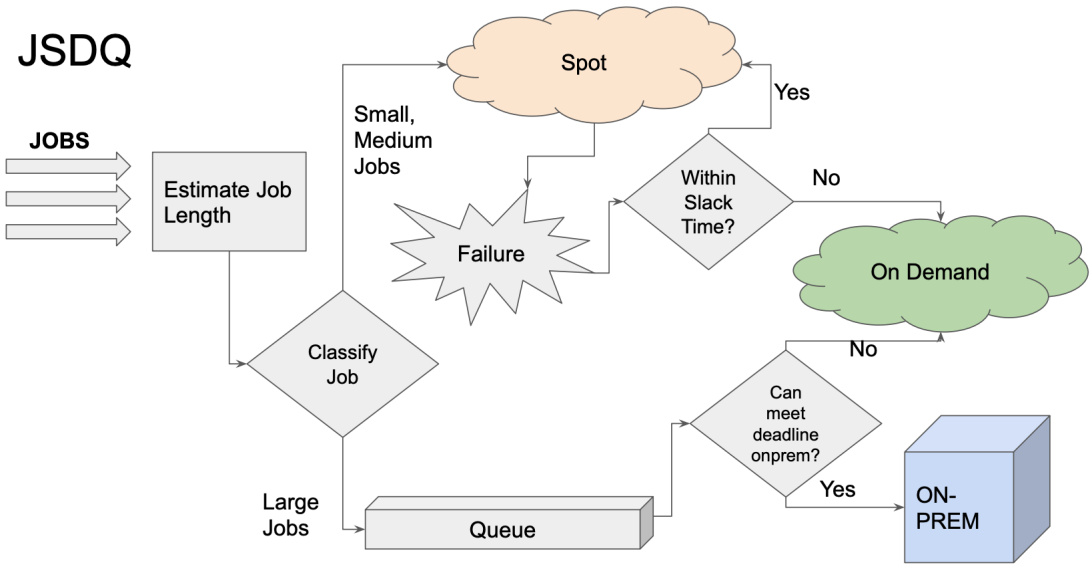


Figure 3.4: Flowchart of JSDQ algorithm. JSDQ utilizes spot instances whenever possible. It proactively calculates onprem queue waiting time and job slack time to determine the possibility of deadline miss, then switch to ondemand as a last resort to ensure deadlines are met.

JSDQ employs two additional techniques to ensure that jobs are finished within the deadlines. First, when spot failure happens (e.g., the cloud provider revokes the spot instances), JSDQ considers whether it is safe to retry jobs on another spot instance by calculating a conservative *slack time* for that job. The *slack time* defines a 'safe period' that jobs failing even at 99% progress can still be retried without risking deadline miss. JSDQ currently uses $2\times$ the job's duration as *slacktime*. Second, for any job which enters onprem queue, JSDQ will calculate its expected waiting time. This is done by keeping track of the total duration of jobs waiting in the queue and the progress of previous jobs executed on onprem machines. If JSDQ estimates that the expected waiting time cannot meet the job deadline, it sends the job to ondemand to ensure the job is completed before the deadline.

CHAPTER 4

SIMULATOR DESIGN

Although evaluating the proposed scheduler on a real hybrid cloud system is ideal, this approach can be exhaustively expensive and risky. This is especially true because GPAS is a production system. Real system implementation requires modifying the implementation of GPAS' cluster manager or building a new one from scratch. Neither option can be done without significant effort. A more reasonable approach is to evaluate the idea using a high-fidelity simulation, which provides a cheap and reliable way to test the core idea of the scheduler before investing in a high-risk endeavor, such as modifying the existing production system [77, 78]. Undeniably, the challenge lies in building a reliable simulator for resembling bioinformatics workflow executions on hybrid cloud environments. The following sections describe how I attempt to achieve this goal.

4.1 Events Definition

We built a discrete-event simulator to reliably and affordably prototype our idea. As a discrete-event simulator, our simulator advance in time according to *events* happening within the simulation. This is in contrast to continuous simulation, which advances its time according to a real clock. The reason for choosing discrete-event rather than continuous simulation is pragmatic. Through trial and error, I found that a continuous simulator is easier to build, but its simulation time heavily depends on the length of the real-life situation we would like to observe. Since the proposed scheduler will be evaluated using a few month's worths of GDC's workload, I perceived continuous simulation would take too much time in the long run. In contrast, a discrete-event simulator is relatively more complex to build since it requires lots of event handling, but the complexity pays off through its ability to "jump" directly to the time important events happen. This leads to a much shorter simulation time.

However, it is important to justify how we define *event* within our simulator. This definition will constrain the phenomena, which can be observed throughout the simulation. Too narrow a definition might cause the simulator to miss phenomenons that might be important and/or insightful. On the other hand, if the definition of events is too broad, the simulator can overwhelm itself with unnecessary information, which will obscure analysis rather than illuminate. More information also translates to more computation during the simulation. Hence, a good simulator should be balanced between these two opposites.

When simulating bioinformatics workflow execution, the following events are essential to determine the performance of a scheduler: *a) Job arrival*, *b) Job submission*, *c) Job completion*, *d) Job failure*, *e) Machine instantiation*, and *f) Machine termination.* Job arrival and its completion counterpart decide the job turnaround time, which is an important metric that determines whether a job satisfies its deadline or not. The two events also force the scheduler to make its next scheduling decision. For example, when a job arrives, the scheduler should decide whether it needs to immediately schedule the job for execution, and if so, which machine should execute it, or whether the job should wait in the queue. The latter usually happens when all machines are occupied (except the scheduler uses a delay-based algorithm). In this case, the completion of a running job might mean the start of scheduling operation for the next job in queue.

Since the proposed scheduler will use transient cloud instances (e.g., AWS spot VMs) to execute jobs, failures are bound to happen. Therefore, it is necessary to record the events to calculate turnaround times correctly. The decision regarding when and where a job is retried depends on the active scheduler's failure policy. Finally, it is important to take into account machine instantiation and termination time. Both events are crucial for accurate cloud cost calculation, which is an important metric for evaluating our proposed scheduler. Moreover, spot instance revocation can also be considered a forced machine termination by the cloud provider. A revoked instance causes the job executing within it to fail. Thus,

Event	Scheduling Decision	Cluster State	Metrics		
			Turnaround Time	Cost	Deadline Miss
Job arrival	✓	-	✓	-	✓
Job submission	-	✓	-	-	-
Job completion	✓	✓	✓	-	✓
Job failure	✓	✓	-	-	-
Machine instantiation	-	✓	-	✓	-
Machine termination	✓	✓	-	✓	-

Table 4.1: Types of event in the simulator and how they affect scheduling decision, cluster state, and simulation metrics.

a proper simulation should trigger the active scheduler's failure policy when an instance is revoked.

4.2 Architecture

To faithfully simulate the events described above and capture their resulting metrics, I built a simulator, composed of 5 major components, namely *trace generator*, *execution time predictor*, *scheduling policy*, *hybrid cluster*, and *event processor*. Figure 4.1 depicts the overall architecture of the simulator. Since *execution time predictor* and *scheduling policy* have been extensively described in Chapter 3, this section will only describe their role in giving input to or receiving output from the other components.

Trace Generator. The trace generator takes GDC's historical workload as input and produces a trace of jobs for the simulator. An input trace is a form of a comma-separated-value (CSV) file. Each row denotes one incoming job, while corresponding columns provide the necessary information to generate events. Table 4.2 shows the fields of a trace.

Often times it is necessary to modify the historical workload in order to evaluate the scheduler's performance on different workload characteristics. For this purpose, the trace generator is equipped with the ability to *cut*, *classify*, *sample*, *rerate*, and *resize* workload. Cutting means taking the jobs during a certain window period. Classify serves to identify jobs within certain percentile of execution time, which is especially useful for the proposed

Trace Field	Description
Arrival Time	Job arrival time
Job UUID	Job Identifier
Workflow UUID	Type of GDC workflow requested for the job
Input Size	Total input size for the job, including both references and read files
Execution Time	Time taken to execute job on GPAS cluster

Table 4.2: A description of each field in the simulator trace.

scheduler design. Sample is useful for reducing the amount of simulated jobs (thus reducing simulation time) without while preserving workload characteristics. Rerate modifies the intensity of the workload by adjusting interarrival time between jobs. Finally, resize multiply the execution time of jobs by a certain factor.

Scheduling Policy. The scheduler[1] is responsible for making two important decisions. i.e., *where* and *when* to place an incoming job. The decision on "*where to place an incoming job*" is to select a specific machine from on-prem resources, on-demand VMs, or spot VMs. " *When to place an incoming job*" is to determine whether the job should be scheduled immediately or put in a queue to be processed later.

The simulator considers a scheduler as an abstract class, where each subclass needs to implement the following interface (set of methods): (1) *send job to waiting queue*, (2) *send job to onprem*, (3) *send job to ondemand*, (4) *send job to spot*, and (5) *retry failed job*. In other words, the simulator considers those five methods as sufficient to evaluate the differences between bioinformatics workflow schedulers. Figure 4.2 shows how the five methods carry out a scheduling decision and submit a job to the hybrid cluster. The scheduler considers three types of jobs during its scheduling operation, namely a new incoming job, a waiting job, and a retry job. The scheduler first inspects the state of the cluster to decide whether it can submit a job. If it is not possible, such as because there are no available machines nor budget to cloudburst the job, it will send the job to the waiting queue. If the submission is possible,

1. The terms *'the scheduler,' 'the active scheduling policy,'* or simply *'the policy'* will be used interchangeably throughout this thesis.

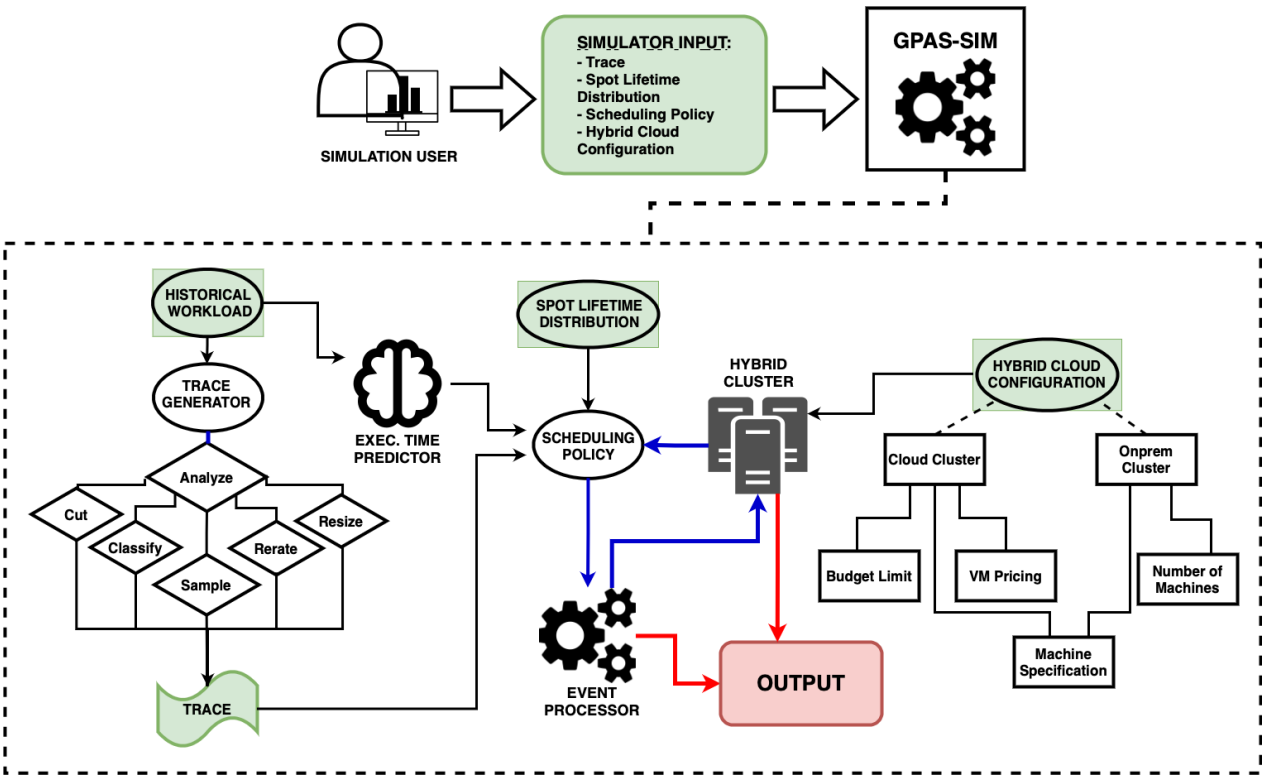


Figure 4.1: Relationships among the most important components of GPAS Simulator. The number of actual components are more than what depicted in the diagram. User-supplied information are denoted with green, while simulator's output is in red.

and the scheduler's algorithm considers that as the best course of action, the scheduler will send the job either to an onprem machine, spot, or ondemand machine. Finally, the event processor (not shown in Figure 4.2) will proceed to modify the hybrid cluster's state accordingly.

Hybrid Cluster. The simulator used an object-oriented approach to model a hybrid cluster. The three most important objects in the model are onprem machines, cloud machines, and the hybrid cluster itself. A hybrid cluster consists of an onprem cluster with a fixed number of machines and an elastic cloud cluster with a variable number of VMs.

Each simulation starts with the hybrid cluster having zero cloud machines and a fixed number of onprem machines as specified by the user. Initially, all machines are idle. As the simulator processes the workload trace, jobs start to arrive, and the active scheduler

will submit the jobs to the cluster. The scheduler is responsible for choosing which machine the job should be executed. If an onprem machine is chosen for the job, the machine will be active (processing the job), and the onprem cluster utilization will increase. If a cloud machine is selected for the job, the scheduler will instantiate either an ondemand or spot VM depending on the scheduler's decision, and thus the capacity of the cloud cluster changes.

On-demand instances will be terminated immediately after their job finishes execution. For spot instances, each machine will be assigned a lifetime based on a distribution given by the user. Spot instances will be terminated either when their job finishes execution or has lived as long as its assigned lifetime (revocation), whichever is earliest. In the latter case (revocation), the event is counted as a job failure, and the corresponding job will be restarted according to the active scheduling policy. Each terminated instance will trigger a cost calculation function which increments cloud cost by *instance lifetime* $\times$ *instance hourly cost*, where instance hourly cost is determined by the user at the beginning of the simulation.

Job Execution Time Predictor. The predictor uses one of the models described in Section 3.1 to estimate the job execution time used by the scheduler. To do so, the predictor initially estimates the execution time of all jobs within a given trace, then stores it in a map where each entry is <Job UUID, Predicted Value>. This approach is possible as Job UUID is unique throughout the historical workload. Then, the scheduler and other objects within the simulator can leverage this map whenever they need job execution time information. As mentioned, this section will refrain from describing how the predictor works in detail, and readers are kindly referred to Section 3.1 for those information.

Event Processor. The event process component encompasses all the logic and helper functions necessary to ensure that all the event handling and changes in simulator objects' states are done correctly. For example, when the scheduler submits a job to an onprem machine, the event processor ensures that the machine's state changes from idle to active and the onprem cluster's utilization increases, or in the case of execution on cloud, the

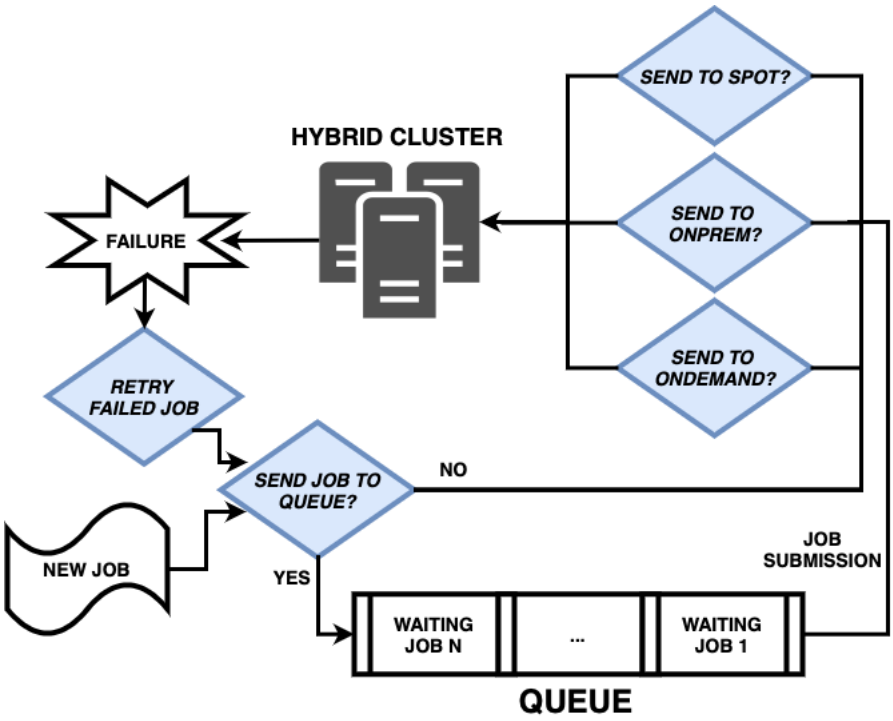


Figure 4.2: The simulator considers a scheduler as an abstract class which implements *Send job to queue*, *Send job to ondemand*, *Send job to onprem*, *Send job to spot*, and *Retry failed job* in order to make a scheduling decision. The five methods are colored in blue.

spot/ondemand VM is instantiated with proper lifetime and cost calculation. Thus, the correctness of the event processor is essential for ensuring the simulator outputs reliable metrics.

Time module is another important component in the event processor. The module is responsible for advancing simulation time according to events happening throughout the simulation. The module does this by providing a parent class *Timeable* to which the hybrid cluster and all machine objects become a child class. Each *Timeable* instance must maintain a *next event time* attribute. For the machine object, its *next event time* value is updated depending on whether a job is submitted, finished, or failed within it. Apart from that, the hybrid cluster object maintains a *current time* attribute which does not exist anywhere else. Then, whenever the event processor has to find the next event, it simply needs to look at two things: the cluster object's *current time* and a list of *next event time* among the cluster's machines. The event processor advances the *current time* to the smallest *next event time*. This approach allows the simulator to cut simulation time by properly advancing only to important events yet maintains a simple logic that is scalable and easily extensible as the simulator gets further developed.

CHAPTER 5

EVALUATION

This chapter presents the evaluation results of JSDQ. All evaluations are conducted using the high-fidelity simulation described in Chapter 4). The chapter proceeds as follows. First, I will describe the environment setup for the evaluation. This setup includes hybrid cluster configuration, the cloudbursting scenarios, the simulation traces I used, and metrics & policies for measuring JSDQ's performance. Following that, I present the evaluation results and briefly discuss how various techniques employed by JSDQ allow it to outperform the other policies.

5.1 Environment Setup

Hybrid Cluster. To evaluate the performance of JSDQ, I simulated a hybrid cluster consisting of 50 identical machines with a variable budget whose value depends on the workload. The budget allows the instantiation of cloud instances during workload spikes. To limit the scope of analysis, this evaluation assumes all machines (both onprem machines and cloud VMs) have identical performance, and their processing rate is constant throughout the simulation. As for the cloud's resource models, this evaluation assumes two types of resources, namely the ondemand and spot instances. Each resource model has its own cost per hour. For spot instance, this evaluation uses \$0.07/hr, which is the hourly price for `c4.2xlarge` instance type at the time of this writing [20]. For ondemand cloud VMs, this evaluation uses a factor of 2.7× *spot instance price*. The factor is reasonable considering ondemand instance pricing [46].

It is important to enumerate the simulation assumptions regarding the hybrid cloud cluster, its machines, and the execution of jobs on the cluster in order to reliably derive insights from simulation results. While a few assumptions are mentioned above, the full list

of assumptions in the simulation is as follows:

(a) We assume no application-level failure happens during job execution. Failures in simulation occur if and only if a job is executed on a spot instance and the job's duration is longer than the spot instance's lifetime.

(b) There are no placement constraints for jobs. The scheduler is free to place any job on any machine without any implication on the job's performance or any of the scheduler's metrics.

(c) Consequently, we assume no resource contention arises from executing jobs on neighboring VMs on the onprem cluster. (GDC onprem cluster, the reference model for our simulated cluster, comprises both physical machines and VMs built on top of neighboring physical servers).

(d) No bin-packing for both onprem and cloud machines, and no DAG-level parallelism for job execution. Thus, all jobs will only occupy one machine throughout its execution until it finishes or fails (whichever is earlier). The occupied machine can not accept other jobs once it starts executing a certain job.

(e) We assume identical performance among all onprem and cloud machines.

(f) No support for checkpointing or job migration during execution. Thus, there exists no post-failure recovery mechanism other than restarting job's execution from the beginning, either on the same or a different machine.

Cloudbursting Scenario. To simulate a cloudbursting scenario, I built upon fundamental principles of queueing theory [79], which describe the relationship between incoming workload and system utilization. Let μ_{dur} be the average duration of jobs arriving at an average interarrival $\mu_{interarr}$. For a system whose machines process only one job at a time, the

average number of occupied machines $N_{occupied}$ at any point in time is determined solely by the fraction of workload duration and interarrival:

$$N_{occupied} = \frac{\mu_{duration}}{\mu_{interarr}} \tag{5.1}$$

In this evaluation, cluster utilization is defined as the number of active onprem machines over the total number of machines. Thus, for any configuration of onprem cluster, if all machines are initially idle, then workload with $\mu_{duration}$ and $\mu_{interarr}$ arrives, Equation-(5.1) also expresses the average utilization of the cluster at any point in time.

It is now possible to devise a cloudbursting scenario for a hybrid cluster with an arbitrary number of onprem machines. Let H be a hybrid cluster with N onprem machines. Let $N_{occupied}(x)$ denote the number of occupied machines at hour x. Then, cluster H is expected to send a portion of incoming jobs to the cloud (i.e., cloudburst them) during hour x if the number of arriving jobs is expected to be bigger than the number of available onprem machines:

$$\frac{1}{\mu_{interarr}} > (1 - N_{occupied}(x)) \tag{5.2}$$

Trace Characteristics. The cloudbursting scenario described above can be simulated using a trace that consists of two parts. The first one is *"non-burst"* part which causes the cluster to be occupied up to a certain utilization level, but not to the point of being required to cloudburst[1]. The second part is *"burst"* part, which represents workload spikes that overwhelm the cluster and force it to cloudburst some portion of incoming jobs. This approach allows us to use different characteristics for the *non-burst* and burst part of the trace. For example, we can design the *non-burst* part such that a third of the cluster is

1. For this evaluation, a hybrid cluster will cloudburst when its utilization is 100%.

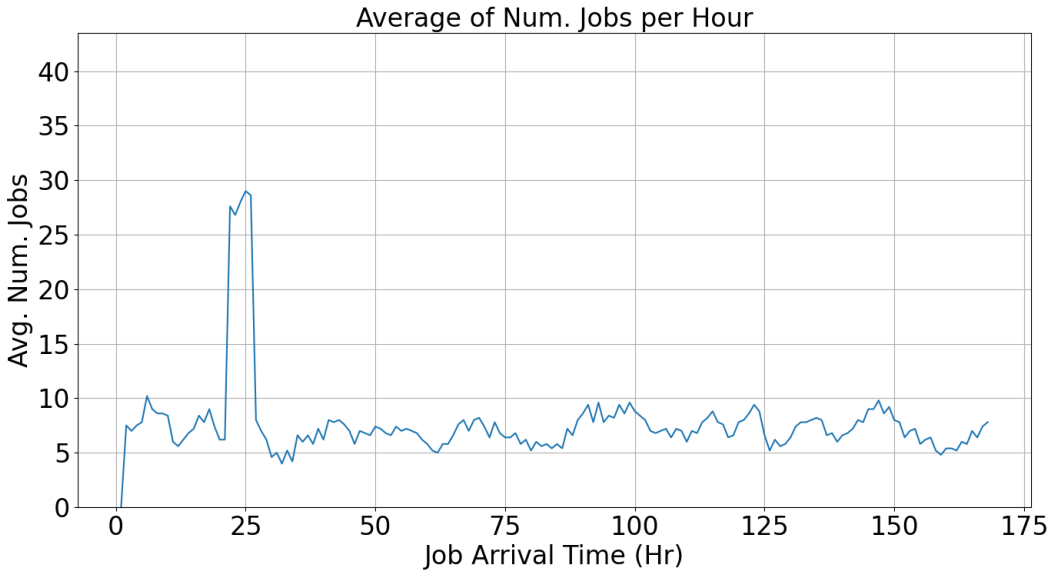


Figure 5.1: Average number of jobs per hour for trace with long jobs burst. A workload spike occurs around $t = 25$.

Non-Burst Characteristics	
Num. Jobs	1194
Total Duration (Hr)	405.62
Avg. Duration (Hr)	0.34
Avg. Interarrival (Hr)	0.14
Latest Arrival Time (Hr)	167.31

Table 5.1: Characteristics of the *non-burst* part of the workload. This will be combined with short and long *burst* to produce two different traces.

utilized, and the *burst* part is composed of a spike of only very small jobs or very long jobs.

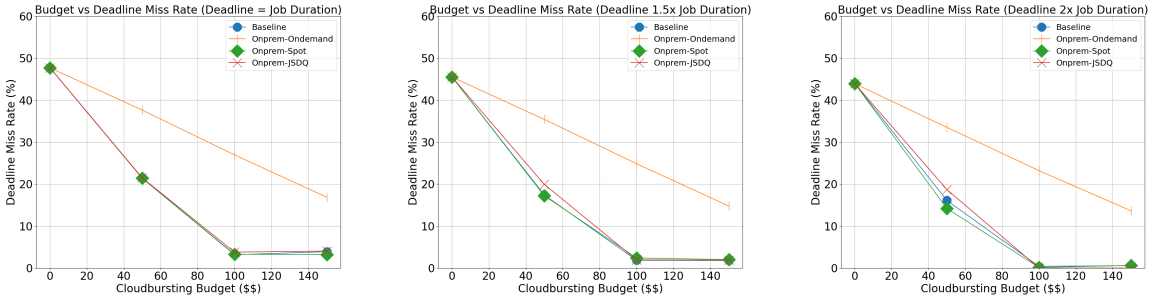
Since JSDQ uses different deadline miss mitigation techniques for small jobs and long jobs, I evaluated JSDQ against spikes of short jobs and long jobs separately. The two scenarios are represented using two different traces, but they have the same *non-burst* part, differing only on the *burst*. For *non-burst* part, I used the Trace Generator described in Section 4.2 to generate workload, which will cause 4% average cluster utilization on the onprem cluster. This *non-burst* part consists of 1194 jobs arriving over roughly 1 week. Table 5.1 shows its characteristics.

Characteristics	***Short Burst***	***Long Burst***
Num. Jobs	811	111
Total Duration (Hr)	1479.7136	4697.9224
Avg. Duration (Hr)	1.8246	42.3236
Avg. Interarrival (Hr)	0.0012	0.0085

Table 5.2: Burst characteristics for short and long jobs. All jobs come within periods t=20 and t=21 (1 hour interval). Short jobs duration range between 0.03 up to 9.96 hours, according to 10 hours *spot threshold* as explained in Section 3.3. Long jobs duration range between 24.92 up to 79.77 hours, which are the 94^{th} and 98^{th} percentile, respectively.

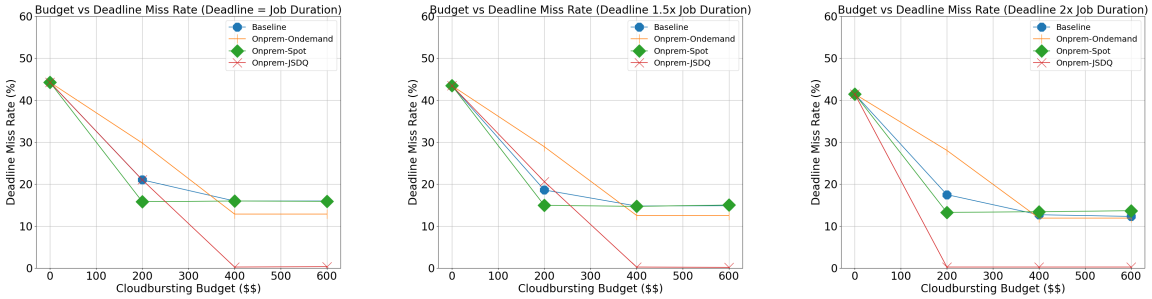
Then, I created two *burst* parts, each for the workload spike of small jobs and long jobs. The small jobs are jobs that are within the $0 - 90^th$ percentile of GPAS dataset. These jobs are expected to trigger JSDQ's usage of spot instances (Section 3.3). The long jobs are within $91^{th} - 98^{th}$ percentile. These should trigger JSDQ's usage of ondemand instances based on queue waiting time. Table 5.2 shows the characteristics of each workload spike type. Finally, each *burst* part is combined with the *non-burst* part, resulting in two traces used throughout this evaluation. Figure 5.1 shows the number of jobs arriving per hour for trace with a burst of long jobs.

Metrics and Baseline Policies. For this evaluation, JSDQ focuses on two metrics; deadline miss rate and cloud cost. I implemented three policies for the performance comparison. The first policy, *Onprem-Ondemand*, is an ondemand-based policy executing *non-burst* part of the workload on onprem cluster and sending *burst* part to the cloud. The second policy, *Onprem-Spot*, is a spot-based policy that employs the same using spot instead of ondemand instances. This policy continuously tries to execute jobs on spot VMs despite repeated failure. Finally, the third policy, denoted *Baseline*, is GDC's way of doing cloudbursting as of this thesis writing. For any incoming job, this policy sends it to spot VMs and allows maximum retry of three times before moving to ondemand instances.



(a) Deadline = Job Duration (b) Deadline = 1.5× Job Duration (c) Deadline = 2× Job Duration

Figure 5.2: Comparison between JSDQ and the other three baseline policies when the burst of small jobs arrives. JSDQ's performance is on par with the spot-based policies (Baseline and Onprem-spot)



(a) Deadline = Job Duration (b) Deadline = 1.5× Job Duration (c) Deadline = 2× Job Duration

Figure 5.3: Comparison between JSDQ and other baseline policies when handling burst of large jobs. JSDQ is able to significantly cut deadline miss rate due to its onprem deadline miss mitigation techniques (Section 3.4).

5.2 Evaluation Results

Based on the current evaluation, JSDQ shows promising results against all the other policies. Figure 5.2 shows JSDQ's performance when handling spikes of short jobs. As shown in the figure, JSDQ provides comparable deadline miss rates to spot-based policies, which all excel in handling short jobs. However, Figure 5.3 shows that JSDQ provides comparable or even better performance than ondemand-based policy when handling long jobs. All spot-based policies perform poorly in this scenario since long jobs cause repeated failures, leading to additional costs for job retry and high turnaround due to wasted computation. Furthermore, in the large jobs scenario compared, JSDQ provides 5% cost savings compared to ondemand-based policy. This can be attributed to JSDQ's queue waiting time prediction that allows it to execute even very long jobs on onprem as long as it does not risk missing the deadline. In contrast, the ondemand-based policy naively sends any job to onprem when there are available machines, without considering whether it will cause a huge waiting time for the next jobs.

CHAPTER 6

CONCLUSION

This thesis presents JSDQ, a novel scheduling algorithm employing preemptible cloud (spot) instances with low availability to execute long-running bioinformatics workflows while ensuring deadline satisfaction. JSDQ leverage spot instances' lifetime distribution to decide a threshold of job duration which guarantees a high probability of successful execution on spot instances, then uses a high-accuracy job execution time predictor to make cloudbursting decision as jobs arrive. In order to evaluate JSDQ, I built a high-fidelity discrete event simulator that allows JSDQ prototyping in a cheap and reliable way. Evaluation using large-scale workload traces from one of the leading genomics research centers shows promising results. In particular, JSDQ is able to provide comparable performance to spot-based policies when handling small jobs, while ondemand-based policy suffers due to its higher cost. On the other hand, JSDQ also provides comparable performance to the ondemand-based policy when processing large jobs, while spot-based policies suffer due to high failure probability. I plan to implement and evaluate JSDQ on one of the leading genomics research center's production systems. I will also further enhance JSDQ by taking into account the DAG nature of bioinformatics jobs, which opens up the possibility of including various DAG scheduling techniques..

REFERENCES

[1] C. S. Greene, J. Tan, M. Ung, J. H. Moore, and C. Cheng. Big Data Bioinformatics. Journal of Cellular Physiology, 229(12):1896–1900, Dec 2014.

[2] Michael Hao Tong, Robert L. Grossman, and Haryadi S. Gunawi. Experiences in Managing the Performance and Reliability of a Large-Scale Genomics Cloud Platform. In 2021 USENIX Annual Technical Conference (USENIX ATC), July 14-16, 2021.

[3] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big Data: Astronomical or Genomical? PLOS Biology, 13(7):e1002195, Jul 2015.

[4] F. Sanger and A. R. Coulson. A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. Journal of Molecular Biology, 94(3):441–448, May 1975.

[5] R. Pereira, J. Oliveira, and M. Sousa. Bioinformatics and Computational Tools for Next-Generation Sequencing Analysis in Clinical Genetics. Journal of Clinical Medicine, 9(1), Jan 2020.

[6] L. Wratten, A. Wilm, and J. Göke. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. Nature Methods, 18(10):1161–1168, 10 2021.

[7] Andrew S. FastQC: a quality control tool for high throughput sequence data, 2010.

[8] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics, 25(14):1754–1760, Jul 2009.

[9] Picard toolkit. `https://broadinstitute.github.io/picard/`, 2019.

[10] Elise Larsonneur, Jonathan Mercier, Nicolas Wiart, Edith Le Floch, Olivier Delhomme, and Vincent Meyer. Evaluating Workflow Management Systems: A Bioinformatics Use Case. In IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Madrid, Spain, December 3-6, 2018.

[11] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. Nature Biotechnology, 35(4):316–319, 04 2017.

[12] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojsa Tijanic, Hervé Ménager, Stian Soiland-Reyes, and Carole A. Goble. Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language. CoRR, abs/2105.07028, 2021.

[13] Z. Zhang, K. Hernandez, J. Savage, S. Li, D. Miller, S. Agrawal, F. Ortuno, L. M. Staudt, A. Heath, and R. L. Grossman. Uniform genomic data analysis in the NCI Genomic Data Commons. Nature Communications, 12(1):1226, 02 2021.

[14] M. Garcia, S. Juhos, M. Larsson, P. I. Olason, M. Martin, J. Eisfeldt, S. DiLorenzo, J. Sandgren, T. Díaz De Ståhl, P. Ewels, V. Wirta, M. Nistér, M. Käller, and B. Nystedt. Sarek: A portable workflow for whole-genome sequencing analysis of germline and somatic variants. F1000Research, 9:63, 2020.

[15] Nikolas Roman Herbst, Samuel Kounev, and Ralf H. Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. In 10th International Conference on Autonomic Computing (ICAC), pages 23–27, San Jose, CA, USA, June 26-28, 2013.

[16] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, André Bauer, Alessandro Vittorio Papadopoulos, Dick H. J. Epema, and Alexandru Iosup. An Experimental Performance Evaluation of Autoscalers for Complex Workflows. ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 3(2):8:1–8:32, 2018.

[17] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Akshay Jajoo, Y. Charlie Hu, Xiaojun Lin, and Nan Deng. A case for task sampling based learning for cluster job scheduling. In Amar Phanishayee and Vyas Sekar, editors, 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022, pages 19–33. USENIX Association, 2022.

[19] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Amazon EC2 Spot Instances Pricing . `https://aws.amazon.com/ec2/spot/pricing/`, Oct. 2022.

[21] Amazon Compute Service Level Agreement . `https://aws.amazon.com/compute/sla/`, 2022.

[22] Spot Instance Interruption Notices . `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html`, Oct. 2022.

[23] Pedro Joaquim, Manuel Bravo, Luís Rodrigues, and Miguel Matos. Hourglass: Leveraging transient resources for time-constrained graph processing in the cloud. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Supreeth Shastri and David E. Irwin. Hotspot: automated server hopping in cloud spot markets. In Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017, pages 493–505. ACM, 2017.

[25] Leo Breiman. Random forests. Machine Learning, 45(1):5–32, oct 2001.

[26] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Elaine R. Mardis. Next-Generation Sequencing Platforms. Annual Review of Analytical Chemistry, 6:287–303, 2013.

[28] Petr Danecek, James K. Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O. Pollard, Andrew Whitwham, Thomas Keane, Shane A. McCarthy, Robert M. Davies, and Heng Li. Twelve years of SAMtools and BCFtools. GigaScience, 10(2):1–4, 2021.

[29] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A DePristo. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. Genome Research, 20(9):1297–1303, 2010.

[30] Dale Muzzey, Eric A Evans, and Caroline Lieber. Understanding the basics of NGS: from mechanism to variant calling. Current Genetic Medicine Reports, 3(4):158–165, 2015.

[31] W. McLaren, B. Pritchard, D. Rios, Y. Chen, P. Flicek, and F. Cunningham. Deriving the consequences of genomic variants with the Ensembl API and SNP Effect Predictor. Bioinformatics, 26(16):2069–2070, Aug 2010.

[32] H. Yang and K. Wang. Genomic variant annotation and prioritization with ANNOVAR and wANNOVAR. Nat Protoc, 10(10):1556–1566, Oct 2015.

[33] M. A. Jensen, V. Ferretti, R. L. Grossman, and L. M. Staudt. The NCI Genomic Data Commons as an engine for precision medicine. Blood, 130(4):453–459, 07 2017.

[34] A. P. Heath, V. Ferretti, S. Agrawal, M. An, J. C. Angelakos, R. Arya, R. Bajari, B. Baqar, J. H. B. Barnowski, J. Burt, A. Catton, B. F. Chan, F. Chu, K. Cullion, T. Davidsen, P. M. Do, C. Dompierre, M. L. Ferguson, M. S. Fitzsimons, M. Ford, M. Fukuma, S. Gaheen, G. L. Ganji, T. I. Garcia, S. S. George, D. S. Gerhard, F. Gerthoffert, F. Gomez, K. Han, K. M. Hernandez, B. Issac, R. Jackson, M. A. Jensen, S. Joshi, A. Kadam, A. Khurana, K. M. J. Kim, V. E. Kraft, S. Li, T. M. Lichtenberg, J. Lodato, L. Lolla, P. Martinov, J. A. Mazzone, D. P. Miller, I. Miller, J. S. Miller, K. Miyauchi, M. W. Murphy, T. Nullet, R. O. Ogwara, F. M. Ortuño, J. Pedrosa, P. L. Pham, M. Y. Popov, J. J. Porter, R. Powell, K. Rademacher, C. P. Reid, S. Rich,

B. Rogel, H. Sahni, J. H. Savage, K. A. Schmitt, T. J. Simmons, J. Sislow, J. Spring, L. Stein, S. Sullivan, Y. Tang, M. Thiagarajan, H. D. Troyer, C. Wang, Z. Wang, B. L. West, A. Wilmer, S. Wilson, K. Wu, W. P. Wysocki, L. Xiang, J. T. Yamada, L. Yang, C. Yu, C. K. Yung, J. C. Zenklusen, J. Zhang, Z. Zhang, Y. Zhao, A. Zubair, L. M. Staudt, and R. L. Grossman. The NCI Genomic Data Commons. Nature Genetics, 53(6):935, Jun 2021.

[35] SLURM Workload Manager . `https://slurm.schedmd.com/`, 2022.

[36] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003), volume 2862 of Lecture Notes in Computer Science, pages 44–60, Seattle, WA, USA, June 24, 2003. Springer.

[37] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In Proceedings of the 11th International Conference on World Wide Web, WWW '02, page 293–304, New York, NY, USA, 2002. Association for Computing Machinery.

[38] Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010), pages 241–252, Indianapolis, Indiana, USA, June 10-11, 2010. ACM.

[39] Sadeka Islam, Srikumar Venugopal, and Anna Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 2015), Kohala Coast, Hawaii, USA, August 27-29, 2015. ACM.

[40] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. ACM Transactions on Computer Systems, 30(4), nov 2012.

[41] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-aware cloud bursting for enterprise applications. ACM Transactions on Internet Technology, 13(3), may 2014.

[42] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '18, page 121–134, New York, NY, USA, 2018. Association for Computing Machinery.

[43] SLURM Cloud Scheduling Guide . `https://slurm.schedmd.com/elastic_computing.html`, 2022.

[44] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 533–546, Boston, MA, July 2018. USENIX Association.

[45] Mark Shifrin, Rami Atar, and Israel Cidon. Optimal scheduling in the hybrid-cloud. In 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013.

[46] Amazon EC2 Instance Types . `https://aws.amazon.com/ec2/instance-types/`, 2022.

[47] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017), Boston, MA, USA, March 27-29, 2017.

[48] Neeraja Jayant Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC 2017), Santa Clara, CA, USA, September 24-27, 2017.

[49] Chin-Jung Hsu, Vivek Nair, Vincent W. Freeh, and Tim Menzies. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018), Vienna, Austria, July 2-6, 2018.

[50] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. Finding the right cloud configuration for analytics clusters. In ACM Symposium on Cloud Computing (SoCC '20), Virtual Event, USA, October 19-21, 2020.

[51] Amazon EC2 On-Demand Pricing . `https://aws.amazon.com/ec2/pricing/on-demand/`, 2022.

[52] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, page 285–300, USA, 2014. USENIX Association.

[53] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In 2017 IEEE International Conference on Big Data (IEEE.

[54] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces. In Proceedings of the International Symposium on Quality of Service (IWQoS 2019), Phoenix, AZ, USA, June 24-25, 2019.

[55] Andrew D. Ferguson, Peter Bodík, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In Proceedings of the Seventh European Conference on Computer Systems (EuroSys 2012), Bern, Switzerland, April 10-13, 2012.

[56] Virajith Jalaparti, Peter Bodík, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015), London, United Kingdom, August, 17-21, 2015.

[57] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), Savannah, GA, USA, November 2-4, 2016.

[58] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011), Boston, MA, USA, March 30 - April 1, 2011.

[59] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016), Santa Clara, CA, USA, March 16-18, 2016.

[60] Yang Hu, Huan Zhou, Cees de Laat, and Zhiming Zhao. ECSched: Efficient Container Scheduling on Heterogeneous Clusters. In 24th European Conference on Parallel Processing (Euro-Par 2018), Turin, Italy,.

[61] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020), Virtual Event, November 4-6, 2020.

[62] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking Beyond GPUs for DNN scheduling on multi-tenant clusters.

[63] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022), Renton, WA, USA, April 4-6, 2022.

[64] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick H. J. Epema. The Grid Workloads Archive. Future Generation Computing Systems, 24(7):672–686, 2008.

[65] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In 19th Annual

IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and TelecommunicationSystems (MASCOTS), Singapore, 25-27 July, 2011.

[66] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In ACM Symposium on Cloud Computing (SoCC 2012), San Jose, CA, USA, October 14-17, 2012.

[67] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the Parallel Workloads Archive. Journal of Parallel Distributed Computing, 74(10):2967–2982, 2014.

[68] Siqi Shen, Vincent van Beek, and Alexandru Iosup. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2015), Shenzhen, China, May 4-7, 2015.

[69] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017), Shanghai, China, October 28-31, 2017.

[70] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In International Conference on High Performance Computing Networking, Storage and Analysis (SC), Seattle, WA, USA, November 12-18, 2011.

[71] Ming Mao and Marty Humphrey. Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows. In 27th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2013), Cambridge, MA, USA, May 20-24, 2013.

[72] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, page 265–278, New York, NY, USA, 2010. Association for Computing Machinery.

[73] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In ECML PKDD Workshop: Languages for Data Mining and Machine Learning, pages 108–122, 2013.

[74] Jacob Benesty, Jingdong Chen, and Yiteng Huang. On the Importance of the Pearson Correlation Coefficient in Noise Reduction. IEEE Transactions on Speech and Audio Processing, 16(4):757–765, 2008.

[75] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[76] AWS Spot Pricing Market . `https://www.kaggle.com/datasets/noqcks/aws-spot-pricing-market`, 2017.

[77] In Kee Kim, Wei Wang, and Marty Humphrey. PICS: A Public IaaS Cloud Simulator. In 2015 IEEE 8th International Conference on Cloud Computing (CLOUD), pages 211–220, 2015.

[78] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In 16th USENIX Conference on File and Storage Technologies (FAST), pages 83–90, Oakland, CA, February 2018. USENIX Association.

[79] Mor Harchol-Balter. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, USA, 1st edition, 2013.

[80] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A. Kozuch, and Gregory R. Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. Technical Report CMU-PDL-16-104, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 2016.