Rochester Institute of Technology

RIT Scholar Works

Theses

7-2019

# Task Scheduling Balancing User Experience and Resource Utilization on Cloud

Sultan Mira
sfm2686@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

Mira, Sultan, "Task Scheduling Balancing User Experience and Resource Utilization on Cloud" (2019). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

# Task Scheduling Balancing User Experience and Resource Utilization on Cloud

by

**Sultan Mira**

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Yi Wang

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

July 2019

The thesis “Task Scheduling Balancing User Experience and Resource Utilization on Cloud” by Sultan Mira has been examined and approved by the following Examination Committee:

Dr. Yi Wang
Assistant Professor
Thesis Committee Chair

Dr. Pradeep Murukannaiah
Assistant Professor

Dr. Christian Newman
Assistant Professor

# Dedication

I dedicate this work to my family, loved ones, and everyone who has helped me get to where I am.

# Abstract


### Task Scheduling Balancing User Experience and Resource Utilization on Cloud

### Sultan Mira

### Supervising Professor: Dr. Yi Wang



Cloud computing has been gaining undeniable popularity over the last few years. Among many techniques enabling cloud computing, task scheduling plays a critical role in both efficient resource utilization for cloud service providers and providing an excellent user experience to the clients. In this study, we proposed a priority cloud task scheduling approach that considers users input to calculate priority, while at the same time, efficiently utilizes available resources. This approach is designed for the consideration of both user satisfaction and utilization of cloud services. In the proposed approach, clients will be required to input their time and cost preferences to determine the priority of each task. We conducted our experiments in Python and AWS to best simulate a real-world cloud environment and compared the proposed approach to a first-come-first-serve approach. We measured the performance of our approach in terms of average task wait time *AWT*, average resource idle time *aRIT*, and the order the tasks were scheduled. The experimental results show that our approach outperforms the first-come-first-serve approach in *AWT*, *aRIT*, and the order the tasks were scheduled.

# Contents

**Dedication** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iii**

**Abstract** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iv**

**1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**2 Related Work** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4**

**3 Methodology** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10**

3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.2 Factors to Be Considered in Task Scheduling . . . . . . . . . . . . . . . . 11
3.3 Task Aging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.4 Calculating Priority Score . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.5 CPU Utilization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.6 Further Considerations . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**4 Experiment** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **16**

4.1 Single Instance Experiment . . . . . . . . . . . . . . . . . . . . . . . . 17
4.1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.1.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2 Multiple Instance Experiment . . . . . . . . . . . . . . . . . . . . . . . 23
4.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2.2 Differences from Single Instance Experiment to Multiple Instance Experiment . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2.3 AWS Instance Specifications and Setup . . . . . . . . . . . . . . . 27
4.2.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**5 Discussion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **31**

5.1 Experiment Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.2 Alternative Solutions . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

5.3 Future Work . . . . . . . . . . 33
5.4 Threats and Limitations . . . . . . . . . . 34

**6 Conclusion . . . . . . . . . . 35**

**Bibliography . . . . . . . . . . 36**

# List of Tables


4.1 Single Instance Experiment Randomly Generated Data-set . . . . . . . . . . 19
4.2 Single Instance Experiment Priority Scheduler Execution Results (Table 4.1 dataset) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.3 Multiple Instance Experiment Randomly Generated Data-set . . . . . . . . . 28
4.4 Multiple Instance Experiment Priority Scheduler Execution Results (Table 4.3 dataset) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

# List of Figures

3.1 Approach Process Flow Overview . . . . . . . . . . . . . . . . . . . . . . 12

4.1 Python Simulation Overview . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.2 Task Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.3 Task Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.4 Custom Priority Queue Implementation . . . . . . . . . . . . . . . . . . . 21
4.5 Randomly Generated Tasks AWT Python Simulation Results . . . . . . . . 24
4.6 Multiple Instance Experiment Overview . . . . . . . . . . . . . . . . . . . 26
4.7 Randomly Generated Tasks Average RIT AWS Multiple Instance Experiment Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

# Chapter 1

# Introduction

There are many numerous reasons why businesses of all sizes, alongside individual users, have started moving to the cloud. Some of those users did it for the sake of the flexibility cloud computing offers, others made the transition hoping to achieve the scalability they need. Whether users were looking to utilize software applications, storage, or just high computing power, they find what they are after on the cloud. Using virtualization technology, cloud providers are able to separate the physical infrastructure and create resources to be consumed and utilized by their users. Because of that cloud computing is able to provide unparalleled services and cater to their customer's individual needs.
Not just any software, or resources, that can be used over the internet can be called a cloud computing system. These five characteristics [15] that a system must have to be considered cloud computing.

1. On-demand self-service: No human interaction required by the consumer to the service provider, or providers, when provisioning capabilities such as processing power, or server time.

2. Network access: The network can be accessed and used by all types of client platforms, such as mobile devices, tablets, and laptops.

3. Rapid elasticity: Computing capabilities are provisioned and released elastically, and even automatically if needed. The resources and capabilities available for use should appear infinite and can be utilized at any given time.

4. Measured service: The usage of resources on the cloud must be monitored and controlled. The reporting of the service and resource usage must be transparent between the provider and the user.

5. Resource pooling: A multi-tenant model where the computing resources of the service provider are pooled to serve more than one user. Generally, users do not possess knowledge or control over the location of the resources they are utilizing. However, users may be able to choose a location for their services at a higher level, such as choosing a state or a country.

Just like any other software, cloud computing is not without issues [21]. One of the most researched issues today in cloud computing is task scheduling. Users submit their tasks to the cloud service provider at any given time and it is up to the cloud provider to execute. Cloud service providers employ scheduling algorithms to ensure proper and efficient resource utilization on their end and attempt to achieve customer satisfaction as well.

Typically, all scheduling algorithms have common goals, regardless of whether it was on a cloud system or not. Those goals include, but are not limited to, resource utilization, throughput, turnaround time, and fairness. Resource utilization enables the cloud service provider to maximize their profits by minimizing the idle time of their resources and always keeping them busy executing customer tasks. Throughput is the measure of how many tasks were processed on an hourly basis. Turnaround time is the amount of time the user of the service spends waiting for the output of their task once they have submitted it to the cloud service provider. The order the incoming tasks are scheduled and the resources they are assigned also has to be fair, based on the criteria the cloud service provider opts to follow.

There is a number of different types of task scheduling algorithms. Perhaps one of the simplest ones is the First Come First Serve (FCFS) scheduling algorithm, whereas the name implies, tasks that are received first, will be scheduled ahead of others. Another type of scheduling algorithms is Shortest Job First (SJF), where the shortest jobs are scheduled first. The size of the task is determined by how much time it will need the CPU to finish

execution. In addition, there is the Priority Scheduling Algorithm where incoming tasks are assigned a priority and are executed on the terms of highest priority first. The calculation of the priority of each task is up to the cloud service provider. Lastly, there is also the Round Robin Scheduling Algorithm where CPU time is broken down to a small unit of time, often called time slice. In Round Robin scheduling, jobs are placed in a circular queue and they are executed for an arbitrary number of time slices until they finished execution.

In this study, we present a priority task scheduling approach that takes into account both cloud service provider's aim at efficient resource utilization and user experience. Cloud computing users depend on the cloud for a verity of different tasks and so those tasks should not be all treated and scheduled using the same scheme. Scheduling algorithms are typically out of the user's control and are completely up to the cloud service provider. We propose an approach that empowers both cloud service providers and their users. The approach enables users to contribute to the decision making of the priority of the tasks they submit to the cloud service provider.

The proposed approach calculates the priority of the incoming tasks to be scheduled based on several factors that users are able to influence. While cloud service providers can consider other factors when scheduling tasks, the factors we considered most important to the users to influence in this study are time and cost. In the proposed approach, users can indicate how fast they would like their tasks scheduled, and at what cost. We ran two experiments to evaluate this approach, a single machine python simulation as well as a seven machine cluster on AWS. We compared the proposed approach to a first-come-first-serve scheduler and the experimental results show our approach outperforming the first-come-first-serve scheduling approach.

The rest of this thesis will go into a number of related studies done on cloud task scheduling, discuss the details of the design of the approach, the setup and results of the experiments conducted to validate the approach and the conclusion of the study.

# Chapter 2

# Related Work

While the work done in ”An application framework for scheduling optimization problems” does not specifically consider cloud environments, it does focus on the process of sequencing and scheduling [10]. According to the authors, there are five different types of scheduling algorithms that could be considered when dealing with a scheduling problem. The five types are: single machine scheduling, job shop scheduling, open shop scheduling, flow shop scheduling, and parallel machine scheduling. The scheduling algorithms proposed by the authors have the goal of achieving optimal performance and results. Choosing which type to choose among those five types will not be an easy task as there are nemours factors to be considered when making that decision.

In works related to cloud resource allocation and management, authors of ”Resource Allocation in the Cloud: From Simulation to Experimental Validation” [14] realize the benefit of of having real-world environments for cloud researchers to test new approaches and algorithms and so they presented testbeds and a testbed adapter on physical clouds. The work and research in their paper makes it possible for cloud research be validated and tested on actual, physical clouds rather than just running simulations. Researchers would need to have extensive knowledge on the underlying cloud platform to run an experiment on a real cloud environment. However, the knowledge many researchers have in the cloud domain may not be adequate to setup a cloud environment to run research experiments. While it might be more cost effective and more convenient to run cloud research experiments in simulated environments instead of real ones, there is always the risk of not considering all of the real-world factors in a simulation.

Thaman and Singh present a review on different scheduling techniques in cloud environments in their paper ”CURRENT PERSPECTIVE IN TASK SCHEDULING TECHNIQUES IN CLOUD COMPUTING: A REVIEW” [19]. The authors state that cloud software is more of a product than a service due to the way it functions and how clients interact with the software. According to Thaman and Singh, quality of service, power usage, privacy and security, VM migration, resource allocation, and task scheduling are all critical existing issues in cloud environments today. In addition, the paper looks into a number of newly invented and tested scheduling approaches such as Metaheuristics, Greedy, Heuristic techniques, and genetics.

In 2010, Selvarani and Sadhasivam proposed a cost-based cloud task scheduling in their paper ”Improved cost-based algorithm for task scheduling in cloud computing” [17]. The scheduling approach they proposed is similar on what factors it considers to the approach we are proposing. The authors of the paper realize that unlike traditional scheduling problems and methods, tasks on the cloud can be associated with different costs. Based on the costs of the submitted tasks, similar tasks are placed in groups and then are scheduled together. The algorithm proposed in the paper takes the cost of both tasks and resource into account when making the task groups.

A study published in 2016 with the title ”Symbiotic Organism Search optimization based task scheduling in cloud computing environment” [1] looked into a new approach for large scale cloud task scheduling. The study proposed the search algorithm Discrete Symbiotic Organism Search (DSOS) and compared it to Particle Swarm Optimization (PSO) algorithm in a simulated environment. According to the authors of the study, (PSO) is a popular heuristic optimization technique that is typically utilized in task scheduling problems. The authors state that their proposed approach performs better as the number of incoming tasks become larger, which makes it a suitable solution for large scale scheduling problems. In addition, the tests the authors ran show DSOS had a superior performance against PSO for large datasets.

Rather than focusing on each task’s CPU and memory requirements, a study published in

2012 focuses on the bandwidth requirements for incoming tasks to propose a new approach for cloud task scheduling [13]. The authors call their proposed algorithm bandwidth-aware task-scheduling (BATS). To validate the performance of the proposed algorithm, CloudSim toolkit was utilized. The authors utilized CloudSim to compare the bandwidth-aware task-scheduling algorithm with few others, including, fair-based task-scheduling and computationonly taskscheduling. The reported results of the algorithm comparison show the bandwidth-aware task-scheduling algorithm to have better performance.

In 2016, Singh and Chana proposed a survey discussing challenges and problems in task scheduling titled: "A Survey on Resource Scheduling in Cloud Computing: Issues and Challenges" [18]. The survey proposed by the authors methodically analyzed 110 papers selected out of a 1206 paper pool published in 19 conferences, symposiums, and workshops, in addition to 11 outstanding journals. The authors state that it is still difficult for researchers to address today's current cloud scheduling problems with the existing resource allocation policies. The authors also stated literature concerning the thirteen types of resource scheduling techniques and algorithms. The study also describes eight different types of resource distribution policies. The main goal of the study, according to the authors, is to aid researchers in choosing resource scheduling algorithms most appropriate to their needs and workloads.

Furthermore, the authors of "Enhanced Particle Swarm Optimization For Task Scheduling In Cloud Computing Environments" [3] state that task scheduling is a key factor for the efficiency of the whole cloud system. The authors define task scheduling as allocating the best suitable resources to execute a task based on different parameters. Those parameters include time, cost, reliability, availability, and resource utilization among others. According to the authors, the majority of task scheduling algorithms do not consider reliability and availability in a cloud environment due to the complexity of achieving them. The study proposes a mathematical model that takes both reliability, availability, execution time, transmission time, make span, round trip time, transmission cost, and load balancing into consideration when scheduling tasks on the cloud. The experiment conducted by the

authors show the proposed approach can save in make span, execution time, round trip time, and transmission cost.

Moreover, in 2014 a paper introducing a proven scheduling framework was publish by the title: ”Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing” [4]. The authors state that it is becoming more challenging and critical at the same time to efficiently schedule tasks over cloud-scale computing as cloud clusters are becoming increasingly large with more diverse characteristics. In this study, the authors introduce Apollo, scalable cloud scheduling framework. Apollo has been deployed on Microsoft’s production clusters scheduling millions of tasks. The authors state that Apollo considers future resource availability when making scheduling decisions and is able to handle unexpected system dynamics.

Keshanchi and Navimipour published a study in 2016 with the title: ”Priority-Based Task Scheduling in the Cloud Systems Using a Memetic Algorithm” [9]. The study introduces a new algorithm to handle cloud priority scheduling. According to the authors, memetic algorithms have been successfully utilized as evolutionary or population-based global search approaches to optimize NP-complete problems. Since task scheduling is also an NP-complete problem, the authors present a new task scheduling algorithm that combines multiple priority queues and a memetic algorithm. Keshanchi and Navimipour validated their algorithm on Azure Cloud Service in C# and compared to three other scheduling algorithms, the results show the proposed algorithm preformed better in terms of make span.

Additionally, Lakra and Yadavb published ”Multi-Objective Tasks Scheduling Algorithm for Cloud Computing Throughput Optimization” [11] in 2015. The authors state that most cloud scheduling algorithms only consider one factor, execution time, so they introduced a new multi-objective task scheduling algorithm that considers more than one factor because they also state that in a cloud environment it is essential to consider various factors. The authors of the study utilize CloudSim as their simulator to validate their proposed approach.

Similar to the scheduling approach we propose in this study, ”Credit Based Scheduling Algorithm in Cloud Computing Environment” [20] proposes a similar approach that attempts

to efficiently utilize available resource and provide excellent user experience. According to the authors of the study, cloud service providers focus more on resource utilization rather than adding more resource to execute user submitted tasks. The authors of the paper analyzed traditional scheduling algorithms and introduced a new improved scheduling algorithm based on task length and user priority.

Focusing on solving the issue of task execution failure on the cloud, ”Fault tolerance aware scheduling technique for cloud computing environment using dynamic clustering algorithm” [12] introduced a new algorithm for cloud scheduling. The authors state that task execution failure is becoming a common occurrence in cloud environment and not much attention has been paid to this issue by the research community. The experiential results show the proposed algorithm greatly reduced task execution failure. The validation of the algorithm included comparing it to the MTCT, MAXMIN, ant colony optimization, and genetic algorithm-based NSGA-II.

Another study focusing on resource utilization published in 2018, ”A New SLA-Aware Load Balancing Method in the Cloud Using an Improved Parallel Task Scheduling Algorithm” [2]. The authors of this study proposed a new parallel genetic algorithm-based method that uses priorities for cloud task scheduling. The main goal of the newly proposed algorithm is to utilize resources efficiently and reduce wasting resources. Validating the algorithm, the authors used Matlab to simulate an environment where they compared their algorithm with two existing techniques, a round-robin based load balancing method, and a hybrid ant colony-honey method. The proposed algorithm in the study performed better than the other two methods in terms of lower energy usage, lower migration rate, and better service level agreement.

Further, in 2018 Gawali and Shinde published ”Task scheduling and resource allocation in cloud computing using a heuristic approach” [6]. The authors combined a number of algorithms and methods to propose a new approach to handle cloud task scheduling and resource allocation. The authors state that each incoming task is processed before it is

allocated to a resource using. the MAHP process. The newly proposed approach combines BATS [13] and BAR optimization methods to consider the bandwidth as a constraint. Turnaround time response time were used as metrics in the experiment conducted by the authors to compare their proposed approach with BATS and improved differential evolution algorithm, IDEA.

There is a number of other studies focusing on the task scheduling on the cloud, for example [16], [5], [7], [22] and [8]; however, their ideas and approaches are quite similar with the literature discussed previously. Thus, we do not cover their details in this section. Our approach, to be presented in the following sections, distinguishes itself from the previously discussed studies by considering the user input and allowing users to contribute to the scheduling of their tasks.

# Chapter 3

# Methodology

In this section, we will go over a brief overview of the proposed approach and then go over the details of each component. This section will also include any assumptions that were made during the design of the approach.

## 3.1 Overview

The approach is intended to work as a task scheduler for cloud systems that is extensible and modifiable. What this study describes serves as guidelines for those who wish to adopt this scheduling approach. One major contribution in this paper is enabling users to have more influence in the scheduling of their tasks to improve their user experience while at the same time maintaining high resource utilization for the provider.

In the proposed approach, users are separate from their tasks so that the same user is able to submit multiple tasks that do not necessarily have the same priority. Each task is treated individually when it comes to its priority. When users submit their tasks for execution, they indicate their time and cost preference for that task and how much resources that task needs. Each task in the system has an individual priority score. For the sake of simplicity, the smallest resource is considered to be 1 CPU, so the scheduler will be expecting integers for task resources. There is no upper limit for how much resources one task can use, the maximum number of resources per task is only constrained by what the provider has available.

When the scheduler receives a new task to be scheduled, that task gets pushed to a priority

queue. The priority queue reorders the tasks using their calculated priority score based on the time and cost factors in descending order. The scheduler will only look for tasks to execute on a specified time interval. The time interval can be determined based on a number of factors such as the frequency of incoming tasks and the total number of resources available. If the scheduler attempts to execute tasks as soon as resources become available, tasks with higher resource needs risk getting ignored and never getting executed. On the specified time interval, the scheduler will invoke the priority queue for the next task to be executed. Once the queue is invoked to return the next task, it will update the age of all the tasks in the queue and then search for the next task. If the task on top of the priority queue can be executed using the resources available to the scheduler at the time, the queue will return it to the scheduler; otherwise, the queue will look for the next highest priority task that can be executed with the available resources. The scheduler will keep requesting tasks to schedule until it runs out of available resources, then the scheduler will do wait until it can execute tasks again.

## 3.2 Factors to Be Considered in Task Scheduling

While each cloud service provider can determine what factors they wish to consider determining task priority, in this study we only considered time and cost. Although the approach we are proposing only considers two factors, more can be added to the implementation of the approach to further improve and fine-tune the priority scheduling such as the number of tasks the user submitted in the past.

The allowed values for both time and cost must be in the range. 0.1-1 inclusive. The higher the value is, the more the user cares about it and vice versa. Using both values of time and cost for each individual task, a score is calculated to determine that task's priority in comparison with other tasks to be scheduled. If more than one task had the same priority score, they will be treated on a first-come-first-serve basis.

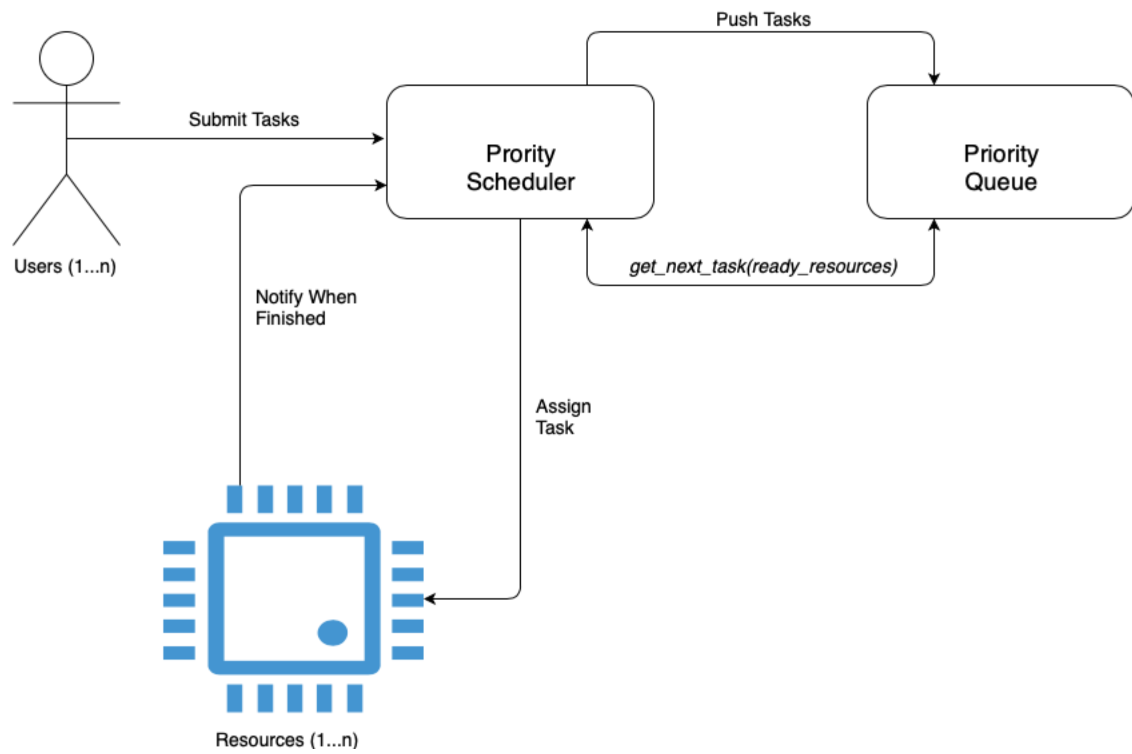


Figure 3.1: Approach Process Flow Overview

## 3.3 Task Aging

Since not all tasks will have the same calculated priority score, we implemented task aging into the proposed approach. The tasks that have lower than average score are at risk of ending up at the bottom of the queue. With incoming tasks having around the average score, or higher than average, tasks at the bottom of the queue will be overlooked because of their low priority and because incoming tasks will consistently have higher priority.

To prevent tasks from being lost and ignored in the queue, we introduced the concept of task aging. When a task object is instantiated, the age for that task starts at 0. The scheduler will update the age of all tasks waiting to be scheduled at the beginning of each scheduling time interval. The age of a task is updated by adding a percentage of the overall task average to their calculated priority score. In practice, this percentage does have to be fixed and can be dynamically calculated, based on incoming task traffic, for instance. The more a task's age is updated, the higher its priority will become, preventing any tasks from being overlooked by the scheduler regardless of their original priority. The calculation for the age of one task is illustrated in formula 3.1, where $p$ represents the percentage chosen to increase the age for task $t$ with $n$ tasks.

$$age_t = age_t + (p \times \frac{\sum_{n=1}^{n} score_t}{n}) \tag{3.1}$$

## 3.4 Calculating Priority Score

The formula is a crucial part of the approach as it takes in the time and cost for each task and outputs the score determining the priority for that task. Because of the context of the priority factors, tasks with high time values indicate that users wish for those tasks to be scheduled quickly, while high cost values indicate users care about cost. So tasks with high time and low cost values get a high priority because it means the user wishes for fast scheduling and they are willing to pay for it. The opposite is also true, tasks with low time and high cost values indicate that the user does not mind delayed scheduling and so those

tasks get low priority.

$$score_t = \frac{2time_t}{cost_t} \tag{3.2}$$

## 3.5 CPU Utilization

The task's priority queue will sort the tasks by their priority on each insertion of a new task and after the age of the tasks in the queue has been updated. Maintaining the order of the tasks enables searching the tasks ready to be scheduled. Searching the tasks becomes useful when the task with the highest priority needs more resources than the scheduler has available at that time. When the scheduler has fewer resources than what the task with the highest priority requires, the queue will search for the next highest priority task that can be executed using the available resources. Searching for the next highest priority task that fits the available resources allows for more resource utilization as it is better to utilize available resources rather than waiting for more resources.

## 3.6 Further Considerations

There are many aspects in the proposed approach that can be changed or modified to the provider's preference and to fit their needs. Some possible modifications are related to the scheduling time interval, priority score formula, priority factors, priority queue, and tasks aging. While this approach is heavily customizable by design, there are few aspects to consider when attempting to customize the approach.

Scheduling time refers to the time the scheduler waits before invoking the queue to schedule tasks. If scheduling time is too short, tasks that require a large number of resources risk getting overlooked because the scheduler never has enough resources to execute them. If scheduling time is too long, tasks will start to pile up in the queue and more importantly, it will lead to poor resource utilization because resources will not be utilized fast enough. So scheduling time is another aspect of the approach that providers will need to determine

based on their traffic and resources.

The formula we propose gives more importance to the time factor, however; that does not always have to be the case. The approach is designed to work with any formula as long as a priority score is generated. There are no constraints on the formula so it is easily modified or replaced to calculate priority scores differently.

Any number of priority factors can be integrated into the approach. In this study, we decided to focus on time and cost to illustrate how the approach would work and because they were the most intuitive factors for us to determine priority. Cloud service providers that wish to adopt this approach can add to the factors we have chosen or replace them completely.

# Chapter 4

# Experiment

This section will go over the experiments we ran on the approach and the results of those experiments. We ran a total of two experiments, one experiment was conducted on a single machine using two processes in Python and the second involved seven instances on AWS also utilizing Python. The goal of both experiments was to serve as a simulation for a real-world environment where tasks are coming in at random with random values and a scheduler using the proposed approach to handle them.

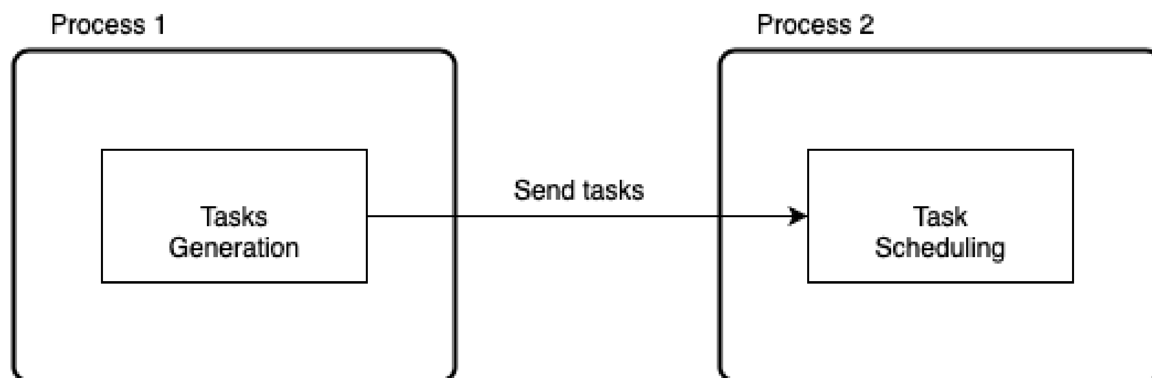


Figure 4.1: Python Simulation Overview

## 4.1 Single Instance Experiment

### 4.1.1 Overview

In this experiment, we ran the proposed scheduling approach on randomly generated tasks with a fixed number of simulated resources and compared the results with a first-come-first-serve scheduling approach. In order to guarantee a valid comparison, the tasks received by both scheduler, although randomly generated, were the same for each execution. In the single instance experiment, the aging increments of the tasks was a fixed value of 1 instead of dynamically calculating it based on the priority score average. The environment and the components both schedulers used were the same, other than the data structure used to store the tasks. We implemented a standard queue data structure with the addition of the methods used in the approach for the first-come-first-serve scheduler while the priority scheduler in the proposed approach used the custom priority queue implementation as its data structure. The version of Python that was used for this simulation was 3.7.1. We only made use of Python's standard libraries and did not use any third-party libraries. There are four main components in the simulation:

1. Task Class

   This class represents a single task in the experiment. The attributes encapsulated in this class are the task ID, time, cost, needed resources, and age. The task class is where the priority score is calculated for each task. The priority score is never stored anywhere and is calculated every time the method is called. The reason for not storing the priority score is to allow the age of the task to influence the score as it increases. If the priority score was stored, then the age of the task will not be considered beyond the initial calculation. Calculating the task priority score repeatedly does not have any negative impacts on performance since the cyclomatic complexity for the score calculation is O(1).

2. Task Generator

```
class Task:

    TIME_WEIGHT    = tw
    COST_WEIGHT    = cw
    AGE_INCREMENT = ai

    def __init__ (self, id, time, cost, cpu):
        self.id    = id
        self.time = time
        self.cost = cost
        self.cpu   = cpu
        self.age   = 0

    def increment_age(self):
        self.age += self.AGE_INCREMENT

    def get_id(self):
        return self.id

    def get_time(self):
        return self.time

    def get_cost(self):
        return self.cost

    def get_cpu(self):
        return self.cpu

    def calculate_score(self):
        return ((self.TIME_WEIGHT * self.time) /
         (self.COST_WEIGHT * self.cost)) + self.age
```

Figure 4.2: Task Class

Table 4.1: Single Instance Experiment Randomly Generated Data-set

| ID | Time | Cost | CPU | Score |
|---|---|---|---|---|
| 0 | 0.52 | 0.78 | 4 | 1.33 |
| 1 | 0.32 | 0.70 | 10 | 0.91 |
| 2 | 0.78 | 0.16 | 9 | 9.75 |
| 3 | 0.57 | 0.18 | 8 | 6.33 |
| 4 | 0.75 | 0.88 | 7 | 1.7 |
| 5 | 0.42 | 0.77 | 2 | 1.09 |
| 6 | 0.81 | 0.44 | 5 | 3.68 |
| 7 | 0.19 | 0.12 | 10 | 3.17 |
| 8 | 0.92 | 0.76 | 7 | 2.42 |
| 9 | 0.35 | 0.48 | 10 | 1.46 |
| 10 | 0.28 | 0.54 | 3 | 1.04 |
| 11 | 0.50 | 0.73 | 1 | 1.37 |

The task generator in this experiment was simulating a collection of end-users submitting tasks to the scheduler. The main job for this component is generating task objects to simulate real tasks being submitted to the scheduler. The generator keeps track of the last task ID that was generated and it increments it by one for each new task. The task generator has a specific time interval to generate the tasks. The time interval is not static as we changed it every time the simulation executed to simulate different situations. For example, to simulate heavy traffic of incoming tasks we gave it the value of 1 second, while for simulating lower traffic it had the value of 8 seconds. The values for time, cost and resources needed for each task were generating using Python's *random* library. The time and cost values were generated to be between 0.1-1 to avoid division by zero and to keep the computation lightweight. The random values for resources needed by each task were generated to be between 1-10. Once a task object is generated, it is then sent to the scheduler using Python's *socket* and *pickle* libraries.

3. Task Scheduler

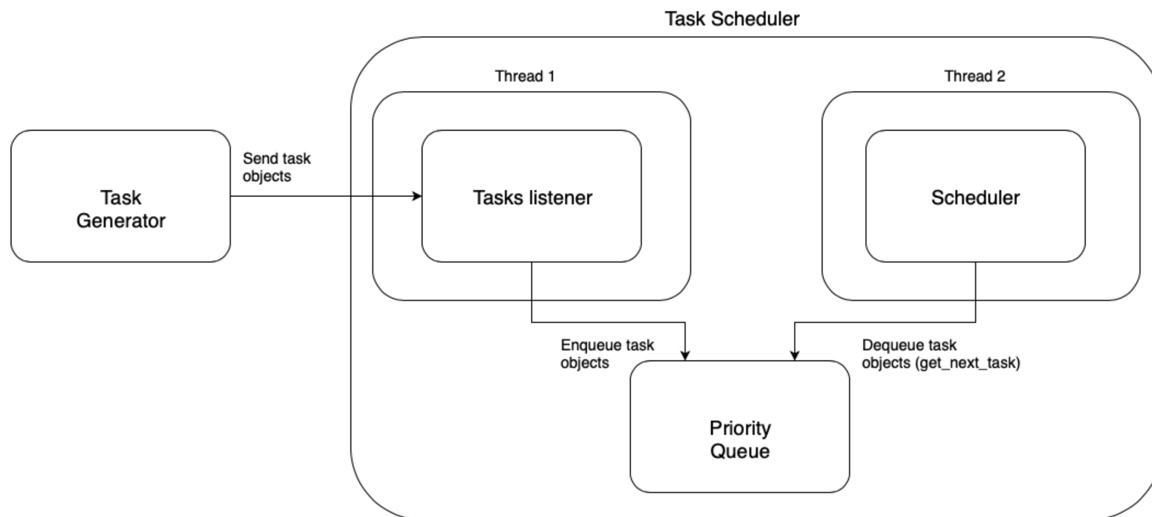


Figure 4.3: Task Scheduler

The task scheduler had the responsibility of receiving tasks and simulating resources. The task scheduler ran on a different process from the task generator to simulate the separation that naturally exists between the end-user and the cloud service provider. In addition, we used Python's *threading* library to run one thread for task scheduling and resource simulation, and another thread to listen for incoming tasks from the task generator process. In this experiment, we simulated 10 CPUs available to the scheduler every 4 seconds. The scheduler will wait for the resources to become available before attempting to invoke the queue for tasks in order to schedule them. Before the scheduler attempts to schedule tasks, it will first signal the queue to update the age of all the tasks waiting to be scheduled by calling *update_tasks_age*. The goal of the scheduler was to fully utilize all of the resources it had available. In order to avoid a race condition, we utilized Python's *threading.Lock* to control access to the shared data between the two threads. The priority queue containing the tasks is the shared data between the two threads as one of them pushed incoming tasks to it, and the other thread scheduled tasks from the queue as can be seen in figure 4.2.

4. Custom Priority Queue

The priority queue sorts task objects based on their calculated priority scores, if two

```
class PriorityQueue:

    def __init__(self):
        self.queue = []

    def is_empty(self):
        return len(self.queue) == 0

    def enqueue(self, item):
        self.queue.append(item)
        self.queue.sort(key=lambda x: x.calculate_score(), reverse=True)

    def update_tasks_age(self):
        for i in range(0, len(self.queue)):
            self.queue[i].increment_age()

    def get_next_task(self, cpu):
        for i in range(0, len(self.queue)):
            if self.queue[i].get_cpu() <= cpu:
                task = self.queue[i]
                # dequeue
                del self.queue[i]
                return task
        return None
```

Figure 4.4: Custom Priority Queue Implementation

tasks had the same priority score, they will be treated on a first-come-first-serve basis. When a task object is pushed to the queue, it is first appended to the underlying list data structure, and then the list is sorted using Python's *sort* function. Unlike a standard queue implementation, we replaced the *dequeue* method with *get_next_task*. The task with the highest priority will not always be fit to be scheduled, *get_next_task* takes in the number of available CPUs to the scheduler as an argument and returns the next highest priority task that can be executed with the available CPUs.

### 4.1.2 Results

We compared the two scheduling approaches in terms of average wait time, *AWT*, per task, and the order the tasks were scheduled. We calculated the average wait time for each of

the two schedulers by first using formula 4.1 to calculate the wait time for each task where *sw* was the time in seconds that the scheduler had to wait for before attempting to schedule tasks. Once we knew the wait time for each task we aggregated the wait times in formula 4.2 to compute the average wait time for each scheduling approach for *n* tasks.

$$wt_t = age_t \times sw \tag{4.1}$$

$$AWT = \frac{\sum_{t=1}^{t} wt_t}{n} \tag{4.2}$$

When we ran the experiment with both scheduling approaches, our proposed priority scheduler and a first-come-first-serve scheduler with the dataset illustrated in Table 4.1. The task generator process sent one task every second to both schedulers by the order of their IDs, starting from task ID 0 to 11. Both schedulers had 10 simulated CPUs available to them every 4 seconds, which was also their scheduling time interval. When both schedulers finished execution, the *AWT* for the priority scheduler was observed to be 10.0, while the first-come-first-serve scheduler had an *AWT* of $15.\overline{66}$. As for the order the schedulers executed the tasks, the first-come-first-serve scheduler executed the tasks in the same order they came in. Table 4.2 illustrates the order our proposed priority scheduler executed the tasks. The tasks were executed in the correct order based on their priority and when they were received by the scheduler.

In addition to using a predefined dataset for the single instance experiment, we also compared both scheduling approaches with dynamically generated tasks and measured the *AWT* at the end of the execution. We fed both schedulers 100, 1000, and 2000 tasks where the generator sent 1 task per second and each scheduler had 25 CPUs available every 4 seconds. The *AWT* with 100 tasks for the priority scheduler was **4.72** while the first-come-first-serve scheduler scored **5.24**. As for the *AWT* of 1000 tasks, the priority scheduler scored **5.108** and the first-come-first-serve scored **7.944**. Lastly, for 2000 tasks, the *AWT* for the priority scheduler was **5.096** and the first-come-first-serve scheduler scored **7.81**. It can be observed in Figure 4.7 how the *AWT* of the two approaches compare and how it starts to diverge as

Table 4.2: Single Instance Experiment Priority Scheduler Execution Results (Table 4.1 dataset)

| ID | Time | Cost | CPU | Score | Age | *wt* |
|---|---|---|---|---|---|---|
| 2 | 0.78 | 0.16 | 9 | 10.75 | 1 | 4 |
| 1 | 0.32 | 0.70 | 10 | 1.91 | 1 | 4 |
| 3 | 0.57 | 0.18 | 8 | 8.33 | 2 | 8 |
| 5 | 0.42 | 0.77 | 2 | 2.09 | 1 | 4 |
| 6 | 0.81 | 0.44 | 5 | 5.68 | 2 | 8 |
| 0 | 0.52 | 0.78 | 4 | 4.33 | 3 | 12 |
| 11 | 0.50 | 0.73 | 1 | 2.37 | 1 | 4 |
| 7 | 0.19 | 0.12 | 10 | 6.17 | 3 | 12 |
| 4 | 0.75 | 0.88 | 7 | 5.70 | 4 | 16 |
| 10 | 0.28 | 0.54 | 3 | 4.04 | 3 | 12 |
| 8 | 0.92 | 0.76 | 7 | 6.42 | 4 | 16 |
| 9 | 0.35 | 0.48 | 10 | 6.46 | 5 | 20 |

the number of tasks becomes larger.

## 4.2 Multiple Instance Experiment

### 4.2.1 Overview

In this experiment, we executed the proposed task scheduling approach in a cloud environment on AWS using 7 total machines. One machine represented the scheduler handling Task object generation and scheduling, the other 6 machines represented resources available to the scheduler. Because we were working with real resources in this experiment, we implemented a simple python sleep script to simulate the resources being utilized. The sleep script takes in an integer as a command-line argument for how many seconds it should sleep the main thread of its process.

We also used Python 3.7.1 as the programming language for this experiment. The communication between the AWS instances was handling using Python's *pickle* and *socket* libraries. We used the same implementation of the custom priority queue in the single

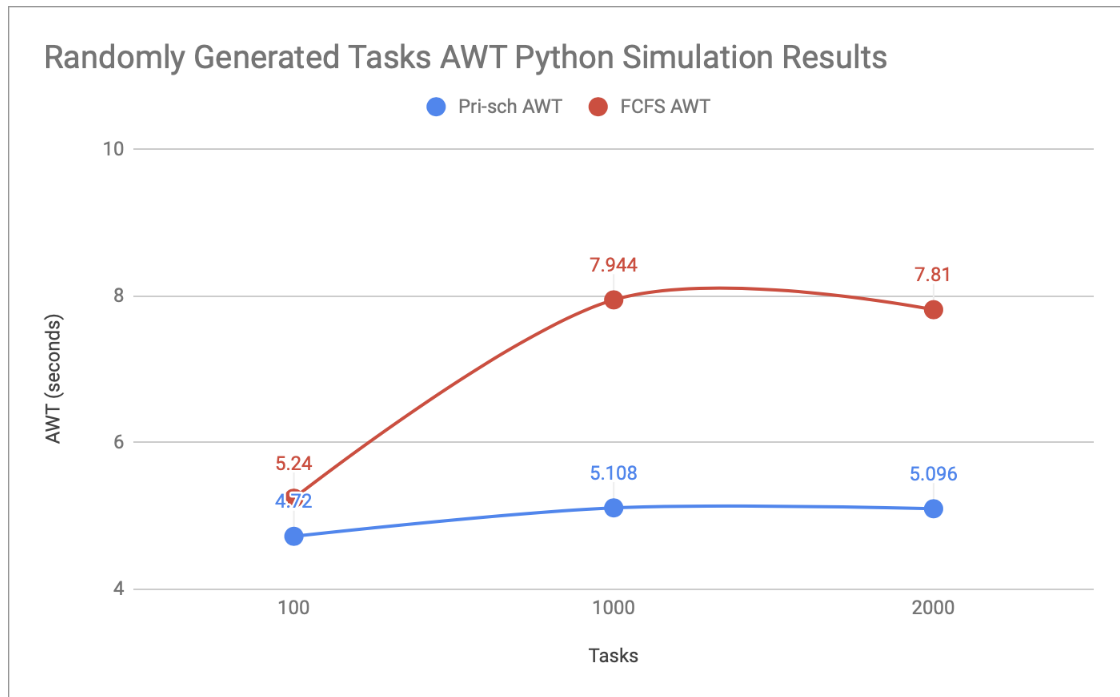


Figure 4.5: Randomly Generated Tasks AWT Python Simulation Results

instance experiment. We aggregated the main components from the single instance experiment into two main components, Task Generation and Scheduling, and Resource Managers. An overview of the entire system can be seen in Figure 4.8.

There were multiple instances of Resource Managers running in the system, we had one instance on each of the resource machines. The Resource Manager component had the job of listening to incoming assignments from the scheduler machine, and notifying the scheduler machine once finished executing the task they were assigned. The incoming assignments in this experiment were a command to run the sleep script along with the integer value indicating how many seconds that resource should not be available for use. We utilized the *subprocess* library in python to execute the sleep script when assigned. Once a resource finished executing the sleep script assigned to it by the scheduler, a signal is sent to the scheduler machine indicating that the resource is ready for another task. Because of the way Resource Managers operated, we only needed the IP address of the scheduler machine to conduct the experiment. Resource Managers handled sending the addresses of each of

the resource machines to the scheduler machine. This way the entire system can be scaled with ease since the Resource Manager component had the responsibility of contacting the scheduler, which means that adding new resources is just a matter of running the Resource Manager component on a newly launched instance.

The second main component in the system was responsible for task objects generation, task scheduling, and keeping track of available resources as they notify the scheduler. To handle those responsibilities smoothly, we implemented 3 concurrent threads using Python's *threading* library. This component maintained two lists, available resources addresses, and the priority queue containing the task objects waiting to be scheduled. The *threading.Lock* library was used to allow the data to be shared between the 3 threads without running into a race condition or a deadlock. The task object generation and the resource listener threads append tasks and resources addresses to the tasks priority queue and the list of resources addresses respectively. The scheduler thread removed tasks and resources from both lists when it assigned tasks to resources.

### 4.2.2 Differences from Single Instance Experiment to Multiple Instance Experiment

For this experiment, we needed to change the Task class. In the single instance experiment, only indicating how many resources each task object needed to execute was adequate because we were working with simulated resources and did not need actual execution. However, we needed to introduce commands to the Task class in the multiple instance experiment because we were assigning tasks to real resources. The Task class implementation in the multiple instance experiment holds an additional attribute representing the commands to execute the tasks.

Moreover, unlike the single instance experiment, the number of resources available in this experiment was finite because we utilized real machines and not simulated CPUs. For that

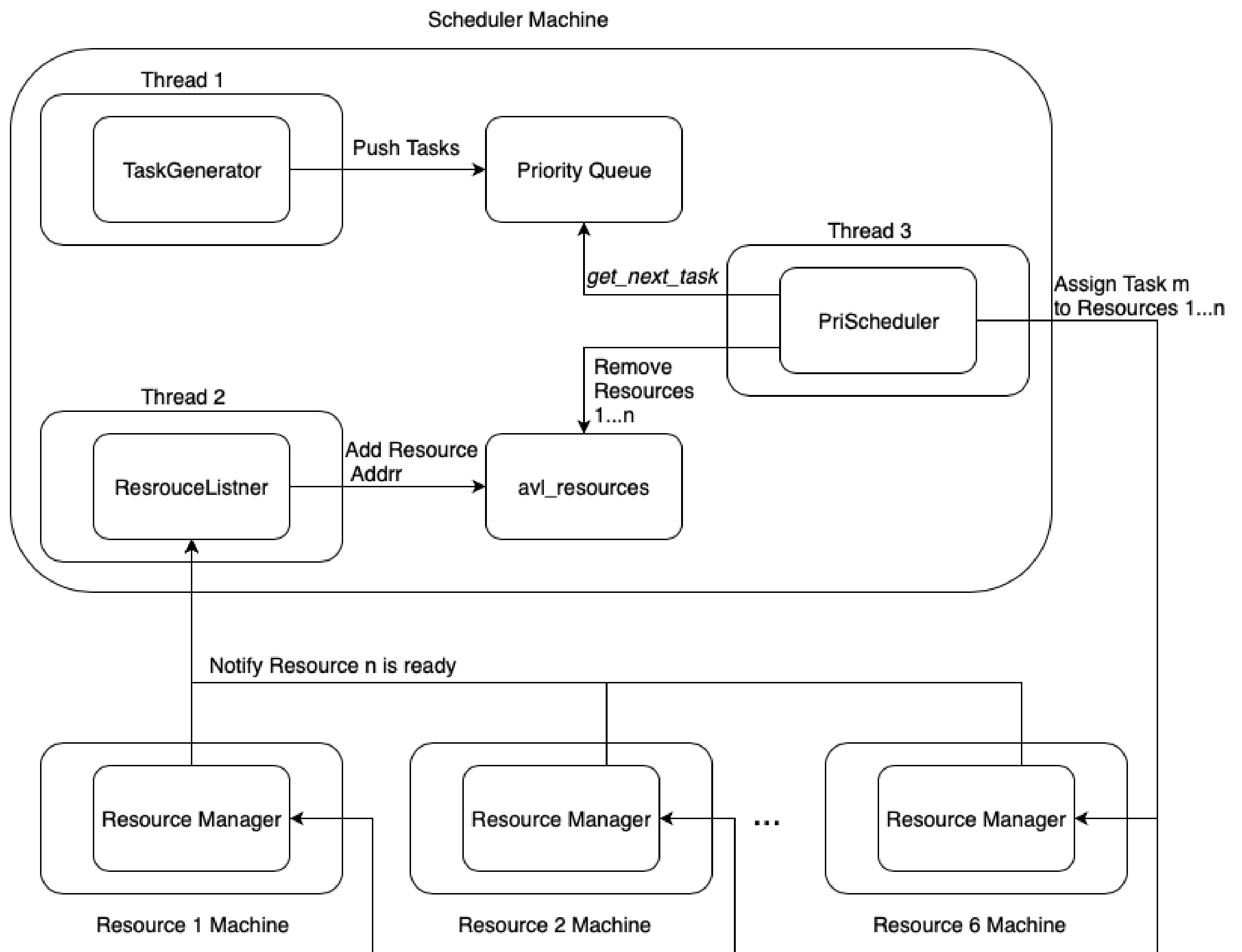


Figure 4.6: Multiple Instance Experiment Overview

reason, we could not compare our proposed scheduling approach with another one simultaneously and we had to run the experiment with each scheduling approach separately. However, to ensure fairness in the comparison, both schedulers get the same tasks, we modified the generation code so it stores the tasks in a *pkl* file using Python's *pickle* library to store objects. We changed the generation thread in the scheduler machine to read the already generated tasks and feed them to the schedulers instead of generating them in real-time.

### 4.2.3 AWS Instance Specifications and Setup

For this experiment we utilized 7 *t2.nano* instances on AWS. The computing power required to conduct our multiple instance experiment was no intensive, which is the reason we choose the *t2.nano* instance instead of the *t2.micro* or *t2.small*. The specifications for each of our 7 instances were 1vCPU, and 0.5GB memory.

Once we had our instances reserved on AWS, we had to modify the security groups configuration in order to allow communication between the different nodes in the experiment. Besides, once we launched the instances and verified that we can establish successful communication among them, we installed our version of Python and cloned our git repository containing the code base for the experiment.

### 4.2.4 Results

We generated 12 random tasks, Table 4.3, using Python's *random* library. The limits for both time and cost values stayed the same in this experiment as they were in the Python one, between 0.1 and 1 inclusive. The fields of *CMD1* and *CMD2* represent the execution of their tasks. For instance, the task with ID 0 requires 2 CPUs, one those CPUs will execute *'python sleep.py 8'* which should occupy it for 8 seconds, and the other required CPU will execute the second command. The number of seconds each *CMD* could occupy a resource for were limited to be between 1 and 20 seconds. Each generated task object had to have at least one *CMD*. Also, due to the limited number of resources in this experiment, the CPU required for each randomly generated task was either 1 or 2 CPUs.

Table 4.3: Multiple Instance Experiment Randomly Generated Data-set

| ID | Time | Cost | CPU | Score | CMD1 | CMD2 |
|---|---|---|---|---|---|---|
| 0 | 0.17 | 0.51 | 2 | 0.67 | *'python sleep.py 9'* | *'python sleep.py 8'* |
| 1 | 0.60 | 0.94 | 2 | 1.28 | *'python sleep.py 16'* | *'python sleep.py 2'* |
| 2 | 0.75 | 1.00 | 1 | 1.50 | *'python sleep.py 15'* | - |
| 3 | 0.23 | 0.63 | 2 | 0.73 | *'python sleep.py 15'* | *'python sleep.py 3'* |
| 4 | 0.86 | 0.78 | 2 | 2.21 | *'python sleep.py 17'* | *'python sleep.py 8'* |
| 5 | 0.26 | 0.68 | 1 | 0.76 | *'python sleep.py 6'* | - |
| 6 | 0.84 | 0.40 | 1 | 4.20 | *'python sleep.py 0'* | - |
| 7 | 0.87 | 0.36 | 1 | 4.83 | *'python sleep.py 10'* | - |
| 8 | 0.86 | 0.86 | 1 | 2.00 | *'python sleep.py 17'* | - |
| 9 | 0.12 | 0.12 | 1 | 2.00 | *'python sleep.py 12'* | - |
| 10 | 0.84 | 0.33 | 2 | 5.09 | *'python sleep.py 9'* | *'python sleep.py 4'* |
| 11 | 0.18 | 0.60 | 1 | 0.60 | *'python sleep.py 3'* | - |

We fed the data in Table 4.3 to two different scheduling approaches, our proposed approach and the first-come-first-serve approach on two separate runs. We measured the *AWT* for both approaches using the same formulas detailed in the results of the single instance experiment, section 4.1.2. We configured the generator to simulate a new task coming into both schedulers every 2 seconds and the schedulers to have a 4 seconds time interval for scheduling attempts. The age increments for the tasks were also 1 in this experiment and was updated once per scheduling attempt, every 4 seconds. After execution of both schedulers with the predefined dataset, the calculated *AWT* was $6.\overline{66}$ for our proposed approach and $6.\overline{33}$ for the first-come-first-serve scheduler. The first-come-first-serve scheduler dispatched the tasks in the order they came in, while the proposed approach's order of dispatch can be seen in Table 4.4.

Moreover, we executed the experiment focusing on resource utilization with larger datasets. We configured our dataset generation to allow each task to require up to 3 CPUs, and to only occupy each CPU for the range of 1-4 seconds. The scheduling time interval for this execution was 2 seconds while the task generation was at 1 second. We ran the experiment after generating 100, 500, and 1000 tasks and measured the average resource idle time,

Table 4.4: Multiple Instance Experiment Priority Scheduler Execution Results (Table 4.3 dataset)

| ID | Time | Cost | CPU | Score | CMD1 | CMD2 |
|---|---|---|---|---|---|---|
| 0 | 0.17 | 0.51 | 2 | 1.67 | *'python sleep.py 9'* | *'python sleep.py 8'* |
| 2 | 0.75 | 1.00 | 1 | 2.50 | *'python sleep.py 15'* | - |
| 1 | 0.60 | 0.94 | 2 | 2.28 | *'python sleep.py 16'* | *'python sleep.py 2'* |
| 4 | 0.86 | 0.78 | 2 | 3.20 | *'python sleep.py 17'* | *'python sleep.py 8'* |
| 6 | 0.84 | 0.40 | 1 | 5.20 | *'python sleep.py 0'* | - |
| 5 | 0.26 | 0.68 | 1 | 1.76 | *'python sleep.py 6'* | - |
| 7 | 0.87 | 0.36 | 1 | 5.83 | *'python sleep.py 10'* | - |
| 10 | 0.84 | 0.33 | 2 | 6.09 | *'python sleep.py 9'* | *'python sleep.py 4'* |
| 8 | 0.86 | 0.86 | 1 | 4.00 | *'python sleep.py 17'* | - |
| 9 | 0.12 | 0.12 | 1 | 4.00 | *'python sleep.py 12'* | - |
| 3 | 0.23 | 0.63 | 2 | 6.73 | *'python sleep.py 15'* | *'python sleep.py 3'* |
| 11 | 0.18 | 0.60 | 1 | 2.60 | *'python sleep.py 3'* | - |

*aRIT*. We added a timestamp to the resource listener thread in scheduler's machine to record when each resource became available. Each time a command was assigned to a resource, that resource's timestamp was subtracted from the current time. For example, if resource *R1* notified the scheduler that it is ready at *(20:20:20)*, that timestamp is recorded along with the resource's address as a Python tuple in the available resources list. At *(20:21:00)*, the scheduler assigned *R1* to execute a command, the time *R1* has spent idle is exactly 1.20 seconds. Since this only calculates the *RIT* for one resource, we utilized the formula illustrated in 4.3 to calculate the average *RIT* for *n* assignments.

$$aRIT = \frac{\sum_{r=1}^{6} RIT_r}{n} \tag{4.3}$$

After running the experiment with the larger datasets, the priority scheduler had an average *RIT* of **1.6677** seconds, and the first-come-first-serve scheduler had **2.2186** seconds for 100 tasks. Additionally, for 500 tasks, the proposed priority scheduler scored an average of **1.5176** seconds and the first-come-first-serve scored **2.0055** seconds. Lastly, for the 1000 tasks dataset, our proposed approach finished execution with an average *RIT* of **1.5164**

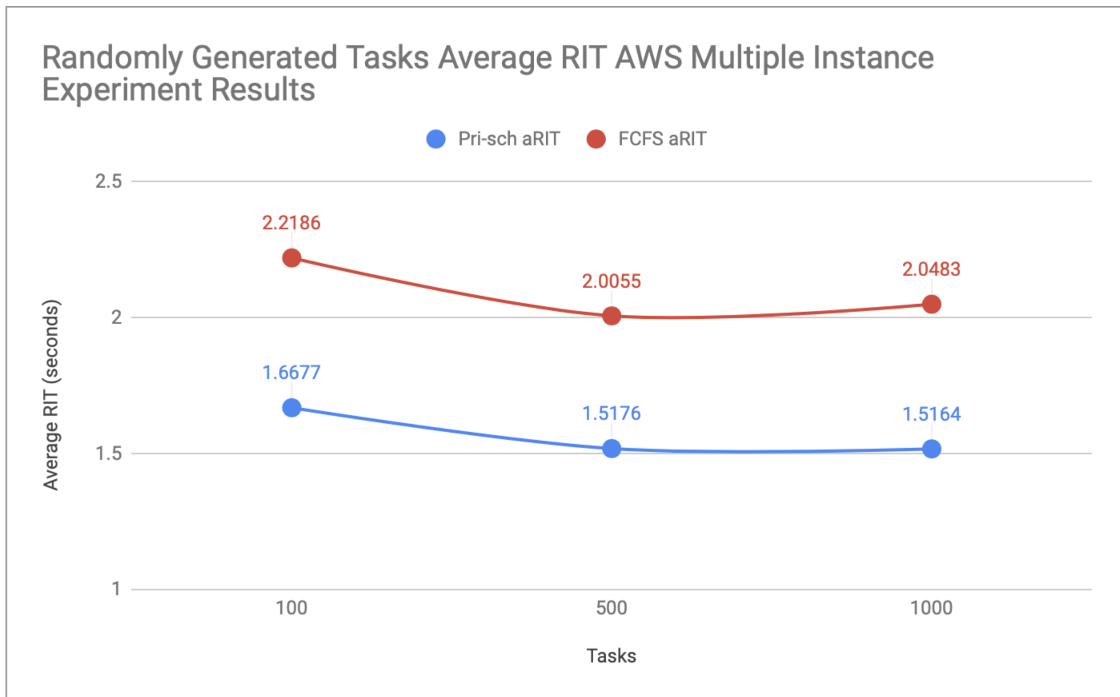


Figure 4.7: Randomly Generated Tasks Average RIT AWS Multiple Instance Experiment Results

seconds and the first-come-first-serve scheduler scored **2.0483** seconds.

# Chapter 5

# Discussion

In this section, we will discuss the reported results of both the single instance and multiple instance experiments. In addition, we will also discuss some alternative solutions to solve the same problem that our proposed approach tackles along with a brief discussion on future work and threats and limitations to the study.

## 5.1 Experiment Results

While having a short *AWT* is good for any scheduling approach, the priority of the tasks take precedence in the proposed approach. It is true that for the predefined dataset the proposed approach scored a better *AWT* than the first-come-first-serve approach did in the single instance experiment, but more importantly, it also performed better when it came to the order of the tasks that were scheduled. Table 4.2 shows what order the proposed scheduler executed the tasks from Table 4.1 along with their final attributes at execution time. When we compared the proposed priority scheduler to the first-come-first-serve scheduler in a simulated heavy task traffic situation and measured *AWT* the proposed scheduler performed consistently better than the first-come-first-serve. Figure 4.7 shows the results of executing the single instance experiment utilize both scheduling approach for 100, 1000, and 2000 tasks. The proposed approach proved it can be scaled to keep up with the demand of heavy incoming task traffic. The *AWT* results calculated for the proposed approach with larger dataset shows the value to be stabilizing as the number of incoming tasks becomes larger. As for the AWS cloud experiment, we measured the *AWT* for a small predefined

dataset, Table 4.3. While both scheduling approaches performed similarly in that regard, the proposed scheduling approach performed better in the order of the scheduled tasks, Table 4.4. Tasks that were supposed to be executed promptly, based on their user's preference, were in fact executed before others, providing better user experience in comparison with the first-come-first-serve scheduling approach.

To validate how scalable the resource utilization efficiency of the proposed approach in a cloud environment we compared the first-come-first-serve scheduling approach to the proposed approach with 100, 500, and 1000 randomly generated tasks. We measured the *aRIT*, average resource idle time at the end of each approach's execution. Figure 4.10 shows how the proposed approach repeatedly had better resource utilization efficiency than the first-come-first-serve approach. The graph in Figure 4.10 shows the tendency for resource utilization of the proposed approach to perform even better as the size of the dataset increases while that tendency is absent in the first-come-first-serve scheduler's results.

## 5.2 Alternative Solutions

There are a number of different solutions to this priority scheduling problem. For instance, dynamic programming could be used in a weighted scheduling algorithm to perhaps provide similar results. The weight of the tasks could be their calculated priority scores and the dynamic programming algorithm would be tasked to schedule tasks giving the priority to the ones that are associated with higher profits. This will ensure appropriate priority scheduling of the incoming tasks and will also attempt to stratifying the user by giving weight to their time and cost preference input.

However, typically for a dynamic programming weighted task scheduling algorithm to work, each task should also be associated with a start and an end time. Adding more information to each task object, while doable, will add more complexity to the solution. The added complexity will not only be in processing more information per task, but also in determining the start and end for each task. Not all tasks are going to be the same and determining the details of how much processing each task requires is not trivial.

## 5.3 Future Work

In the future, we plan to validate the approach after adding the capability of considering a larger number of priority factors simultaneously. A considerable part of adding more priority factors is in the formula which also has the potential to be modified in the future. We plan to modify the formula so it is more complex and supports more priority factors that the ones we proposed at this time. Increasing the number of priority factors and modifying the formula means more accurate priority scores for the tasks and therefore, more effective priority scheduling and more efficient resource utilization. Adding more priority factors will also allow cloud service providers to further customize the proposed scheduling approach to better fit their needs. Some priority factors that we did not consider at this time but might consider in the future include user preference on the geolocation of execution and user task submission history.

In addition, we also plan to modify the generation of the tasks. The task generation we implemented to validate the proposed approach did not follow a specific distribution. We plan to modify the tasks generation in the future to follow a set distribution, in terms of task priory scores, such as the normal distribution. Following different statistical distributions to generate tasks could help simulate different contexts as well, such as heavy traffic of incoming high priority tasks or the opposite. Some cloud service providers might benefit from following a set distribution to generate the tasks as that will show how the approach will handle those tasks.

Furthermore, the concept of the task aging could be replaced by a task timeout policy where the original priority of the tasks is not modified. In a task timeout policy tasks are allowed a certain time in the priority queue, once a task has timed out and stayed in the queue for too long, it has to be scheduled next. This could result in simpler priority score calculations and less required processing power by the approach to operate which is worth considering in the future.

## 5.4 Threats and Limitations

While we performed two experiments in this study to validate the proposed approach, it is still not a replacement for validating in a real-world environment. There were a number of parameters in both of the experiments we discussed that we changed to mimic different real-world situations. There is the threat of edge cases we missed or parameters we configured incorrectly as the approach has only been tested in an environment we controlled entirely. Placing the proposed approach under test in a real-world environment with real tasks being submitted by actual end-users could give us a better understanding of how the user experience could improve to satisfy users better.

In addition, the AWS experiment was conducted with large datasets to validate scalability, however, it was not tested with larger numbers of resources. This poses the danger of cases that were not accounted for when it comes to the approach's scalability. Validating the approach with heavy incoming traffic and allow it access to more resources has the potential of uncovering facts that may have been missed or overlooked regarding the approach. Moreover, the system parameters configured, such as task generation and scheduling time intervals, for the large scale AWS experiment executions were constrained by how many resources were available.

# Chapter 6

# Conclusion

To conclude, we proposed a new priority task scheduling approach on the cloud that strives to achieve excellent resource utilization for cloud service providers and seamless user experience for the end-users. The proposed approach utilizes the end-users time and cost preference input to calculate the priority of each task individually. All the tasks waiting to scheduled are placed in a proposed implementation of a priority queue where they are ordered by their calculate priority scores. The scheduler then invokes the queue to retrieve the next highest priority task that can be executed with the resources the scheduler has available. The approach attempts at scheduling as many tasks as possible at scheduling time to efficiently utilize the resources.

Moreover, we also placed the proposed approach under test in a simulated cloud environment programmed in Python and compared it to a first-come-first-serve scheduling approach. We executed both schedulers with randomly generated tasks and measured the average wait time, *AWT* and observed the scheduling order for each scheduler. The proposed approach performed better than the first-come-first-serve scheduler by having less *AWT* and dispatching the tasks in the correct order of their priority score. Furthermore, we evaluated the proposed approach against the first-come-first-serve scheduler on AWS utilizing 7 machines, where one machine represented the scheduler and the other 6 represented resources. For the AWS cloud experiment, we shifted our focus on measuring the average resource idle time during the execution of each of the two schedulers, *aRIT*. The reported results showed the proposed approach to perform better and will continue to perform well if the size of the dataset increases.

# Bibliography

[1] Mohammed Abdullahi, Md Asri Ngadi, et al. Symbiotic organism search optimization based task scheduling in cloud computing environment. *Future Generation Computer Systems*, 56:640–650, 2016.

[2] Mehran Ashouraei, Seyed Nima Khezr, Rachid Benlamri, and Nima Jafari Navimipour. A new sla-aware load balancing method in the cloud using an improved parallel task scheduling algorithm. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 71–76. IEEE, 2018.

[3] AI Awad, NA El-Hefnawy, and HM Abdel_kader. Enhanced particle swarm optimization for task scheduling in cloud computing environments. *Procedia Computer Science*, 65:920–929, 2015.

[4] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 285–300, 2014.

[5] Yiqiu Fang, Fei Wang, and Junwei Ge. A task scheduling algorithm based on load balancing in cloud computing. In *International Conference on Web Information Systems and Mining*, pages 271–277. Springer, 2010.

[6] Mahendra Bhatu Gawali and Subhash K Shinde. Task scheduling and resource allocation in cloud computing using a heuristic approach. *Journal of Cloud Computing*, 7(1):4, 2018.

[7] Lizheng Guo, Shuguang Zhao, Shigen Shen, and Changyuan Jiang. Task scheduling optimization in cloud computing based on heuristic algorithm. *Journal of networks*, 7(3):547, 2012.

[8] Shaminder Kaur and Amandeep Verma. An efficient approach to genetic algorithm for task scheduling in cloud computing environment. *International Journal of Information Technology and Computer Science (IJITCS)*, 4(10):74, 2012.

[9] Bahman Keshanchi and Nima Jafari Navimipour. Priority-based task scheduling in the cloud systems using a memetic algorithm. *Journal of Circuits, Systems and Computers*, 25(10):1650119, 2016.

[10] Oğuzhan Kocatepe. An application framework for scheduling optimization problems. In *2014 IEEE 8th International Conference on Application of Information and Communication Technologies (AICT)*, pages 1–4. IEEE, 2014.

[11] Atul Vikas Lakra and Dharmendra Kumar Yadav. Multi-objective tasks scheduling algorithm for cloud computing throughput optimization. *Procedia Computer Science*, 48:107–113, 2015.

[12] Muhammad Shafie Abd Latiff, Syed Hamid Hussain Madni, Mohammed Abdullahi, et al. Fault tolerance aware scheduling technique for cloud computing environment using dynamic clustering algorithm. *Neural Computing and Applications*, 29(1):279–293, 2018.

[13] Weiwei Lin, Chen Liang, James Z Wang, and Rajkumar Buyya. Bandwidth-aware divisible task scheduling for cloud computing. *Software: Practice and Experience*, 44(2):163–174, 2014.

[14] Pieter-Jan Maenhaut, Hendrik Moens, Bruno Volckaert, Veerle Ongenae, and Filip De Turck. Resource allocation in the cloud: From simulation to experimental validation. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 701–704. IEEE, 2017.

[15] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[16] Suraj Pandey, Linlin Wu, Siddeswara Mayura Guru, and Rajkumar Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 400–407. IEEE, 2010.

[17] S Selvarani and G Sudha Sadhasivam. Improved cost-based algorithm for task scheduling in cloud computing. In *2010 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–5. IEEE, 2010.

[18] Sukhpal Singh and Inderveer Chana. A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of grid computing*, 14(2):217–264, 2016.

[19] Jyoti Thaman and Manpreet Singh. Current perspective in task scheduling techniques in cloud computing: A review. *International Journal in Foundations of Computer Science & Technology*, 6(1):65–85, 2016.

[20] Antony Thomas, G Krishnalal, and VP Jagathy Raj. Credit based scheduling algorithm in cloud computing environment. *Procedia Computer Science*, 46:913–920, 2015.

[21] Chun-Wei Tsai and Joel JPC Rodrigues. Metaheuristic scheduling for cloud: A survey. *IEEE Systems Journal*, 8(1):279–291, 2013.

[22] Chenhong Zhao, Shanshan Zhang, Qingfeng Liu, Jian Xie, and Jicheng Hu. Independent tasks scheduling based on genetic algorithm in cloud computing. In *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–4. IEEE, 2009.