National College of Ireland

Optimizing Resource Scheduling in Cloud Environments with Docker Containers and Advanced Auto-Scaling Algorithms

MSc Research Project
Cloud Computing

Kurian George
Student ID: X22191437

School of Computing
National College of Ireland

Supervisor: Punit Gupta

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Kurian George
Student ID:	x22191437
Programme:	MSCCLOUD
Year:	2023-2024
Module:	MSc Research Project
Supervisor:	Punit Gupta
Submission Due Date:	12/08/2024
Project Title:	Optimizing Resource Scheduling in Cloud Environments with Docker Containers and Advanced Auto-Scaling Algorithms
Word Count:	XXX
Page Count:	19

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Optimizing Resource Scheduling in Cloud Environments with Docker Containers and Advanced Auto-Scaling Algorithms

Kurian George
X22191437

Abstract

This research presents a novel approach to optimizing resource scheduling in cloud environments by integrating Docker containers with an advanced Ant Colony Optimization (ACO) algorithm. The newly proposed version of the ACO algorithm addresses the inefficiencies of traditional resource scheduling methods by cost-efficiently handling dynamic and fluctuating workloads in cloud infrastructure. This proposed algorithm features adaptive pheromone updating strategies and an improved decision-making process to allocate resources more effectively while minimizing operational costs. This study uses the EdgeSimPy simulator, in which the servers are made as Docker containers for an effective testing environment. This study demonstrates that the proposed new ACO algorithm outperforms conventional methods like First Come First Serve (FCFS) and offers a better and more cost-efficient solution for managing cloud resources. This study marks a significant contribution to cloud resource management by providing an optimal resource scheduling solution in variable cloud environments.

1 Introduction

In this technological era, the use of web applications has skyrocketed and become a daily routine. Thousands of web applications are deployed on the internet daily and are accessed by millions of users around the world. These deployed applications experience varied numbers of users throughout the year. Therefore, depending on the user requests, the resources for the application need to be scaled. Traditionally, these requirements were met by purchasing more servers whenever there was a necessity. However, it is an expensive method, and once the requirement or the peak demand is over, these additionally bought servers become useless and result in a wastage of investment.

Lately, people have started moving to Cloud Platforms to deploy web applications to address the on-demand resource requests. Cloud Platforms offer a pay-as-you-go model, which allows the users to pay only for the resources they use. This feature of the Cloud Platform allowed many developers to host their web applications in the Cloud. This approach eliminates the need to buy additional servers when there is a spike in the application requests than the usual amount, which would otherwise require additional resources to provide quality service to the user Mireslami et al. (2015). A significant amount of revenue can be saved since these additional resources are needed only for a

specific time and can be dynamically released when the requirement is over. Therefore, unlike the traditional methods, a lot of money can be saved by not purchasing new servers.

One efficient way to host web applications on the cloud platform is to containerize and deploy the application. Containerization is the process that allows an application and its dependencies (application's code, libraries, and runtime environment) to run in isolated environments called containers, which share the kernel of the host Operating System (OS) while maintaining isolated user spaces. Containers are efficient and less resource intensive compared to Virtual Machines (VM), as VMs require a separate OS for each instance Srirama et al. (2020). The structure of VMs and containers is shown in figure1. This feature of Containers leads to faster startup times, and better resource utilization, and provides a higher level of security by isolating applications from each other.

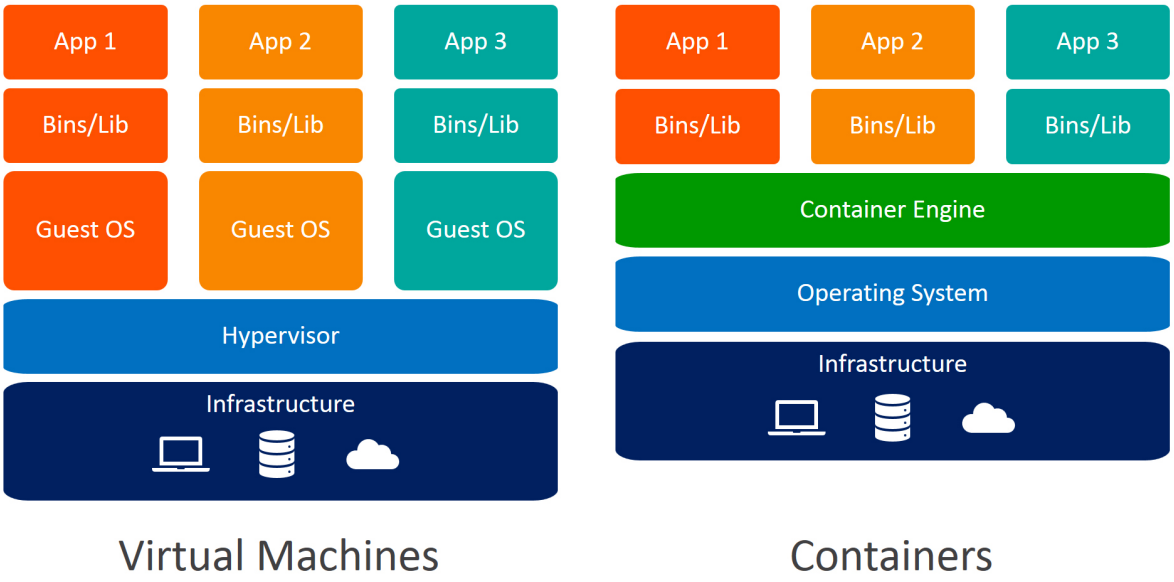

Figure 1: Virtual Machine and Container

There are several methods to containerize web applications, and one such efficient way is by using Docker. Docker is an open-source platform that simplifies and automates the deployment of applications inside containers by using lightweight, standalone container images and dependency packages Wan et al. (2018). It is platform-independent, and the dockerized application can be deployed effortlessly without worrying about the underlying OS. Therefore contributing to an efficient cloud resource scheduling.

The basis of this research is the question "Can we achieve cost-efficient resource scheduling in Cloud Environments using Docker containers and the advanced Auto Scaling algorithm Ant Colony Optimization (ACO)?"

This research focuses on addressing this question. Several objectives need to be satisfied to achieve the solution. Firstly, finding a cloud simulator that can develop the same scenario, as developing and testing this resource scheduling experiment in the cloud is a costly process that requires significant resources. Develop and implement a frame-

work that can run the ACO algorithm to schedule resources. Additionally, compare the results with traditional scheduling algorithms like First Come First Serve (FCFS) to determine the cost efficiency of the proposed model. This research aims to find an effective cost-efficient solution for the proposed resource scheduling problem.

The remainder of the paper is structured as follows: Section 2 is the Related Work, which discusses the previous works, their contributions, and limitations on conventional resource scheduling algorithms, advanced scheduling algorithms, and cloud simulators. Section 3 is the Methodology, which describes the materials, and equipment used in the research and why they are used for this research. Section 4 is the Design Specification, which shows the techniques, and framework used for this project. It also contains the flowchart of the proposed algorithm. Section 5 is the Implementation, which describes in detail how the project was implemented and the outputs generated. The Section 6 is the Evaluation section, where the outputs obtained are discussed in detail and the Section 7 is the Conclusion and Future Work, which summarizes the research's objectives and findings, and also suggests the future works that can be done.

2 Related Work

2.1 Conventional Resource Scheduling Algorithms

The paper by Tawfeek et al. (2013) examines the limitations of conventional scheduling algorithms like Round Robin (RR) and First Come First Serve (FCFS) in cloud computing environments and highlights the advantages of an Ant Colony Optimization (ACO) based approach. The RR and FCFS are commonly used due to their easiness and simplicity of implementation. However, there are major drawbacks to these models. The RR model allocates tasks to the Resource that is the Virtual Machine (VM) in a cyclic order without considering the length of the task and the Resource (VM) capabilities, which leads to resource wastage and increased makespan, especially when there are heavy workloads.

Mishra and Jaiswal (2012) also highlights the bottlenecks caused by the FCFS algorithms and the advantage of using ACO to minimize these drawbacks. The FCFS does task scheduling based on the request arrival time thus completely ignoring the resource requirements and execution times, resulting in the overloading of VMs and inefficient task handling. Whereas Sharma et al. (2022) compares the ACO with advanced algorithms like genetic algorithms (GA) and particle swarm optimization (PSO). However, ACO consistently outperforms them regarding convergence speed and solution quality thus ensuring a better QoS-based task scheduling in cloud computing. These conventional methods which used to work efficiently in the olden days are overthrown by new algorithms In this project ACO algorithm is implemented to efficiently allocate the tasks to the resources.

2.2 Advanced Scheduling Algorithm ACO

The concept of ACO was introduced by French entomologist Pierre-Paul Grasse, he noticed that ants communicate indirectly through chemicals called pheromones. This form of communication involves several factors like modifying environmental factors and can only be accessed by nearby insects of the same species. The first developed algorithm inspired by this ant behavior was Ant SystemsDorigo et al. (1996). However, despite its initial success, Ant Systems could not compete with other algorithms for the Travelling Salesman Problem Goyal (2010). However, the concepts initiated by the Ant Systems were well rooted which lead to additional studies on the ACO. Currently, there are many applications that employ ACO and provide state of the art solution for several optimization tasks Dorigo and Stützle (2019).

The paper by Mishra and Jaiswal (2012) focuses on discussing the application of Ant Colony Optimization (ACO) technique to solve load-balancing issues in cloud environment. As compared to the Traditional approach, ACO approach is more dynamic in nature and hence can be well deployed in the dynamic cloud environment where the workload and resources are very dynamic or keep on fluctuating. ACO is a bio-match algorithm that tries to replicate the nature of ants searching for the shortest routes Stützle et al. (2011). The ants stated in the algorithm outline solutions for the problem; when they traverse the potential solution space, they're said to deposit pheromones on the paths they traverse. These are chemical signals that those other ants follow, and a stronger trail of pheromones means that it is a better run route.

As noted in the studying paper by Sharma et al. (2022), such changes reflect the possibility of the ACO model learning from the situation by updating the pheromone trail maps which are representative of the learning capability of the ant agents. This leads to an increase in load distribution, decrease in the time taken to execute the task, and efficient use of resources as compared to other approaches Dorigo (2007). This way, ACO outperforms RR and FCFS by providing a flexible, adaptive and efficient solution to social network scheduling problems. Therefore, ACO algorithm usage leads to better load distribution, shorter computation times, and optimal usage of resources, which are paramount in cloud services' performance and dependability. '

Li et al. (2011) discusses an advanced task-scheduling algorithm for cloud computing environments. Many variations of ACO algorithms have been released in the past years but each version is specifically designed for certain individual purposes. Therefore, this paper uses the Ant Colony Optimization Regression algorithm to get the desired output. That is to allocate the task based on its size to the available resource which is if the task is small then it should be allocated to the nearest small empty resource and if the task is comparatively huge then it will be allocated to the empty bigger resource available for efficient task scheduling.

2.3 Cloud Simulators

The paper "Ahmed and Sabyasachi (2014)" by Arif Ahmed presents an in-depth analysis of various cloud computing simulators. The Simulators provide a cost-effective, repeatable, and controllable environment to test and model multiple cloud applications and configurations. Some common cloud simulators are CloudSim, the most popular sim-

ulator CloudAnalyst, which has a GUI interface, GreenCloud simulator which focuses on energy consumption analysis, MDCSim which focuses on multi-tier data centers and EdgeAISim for simulating AI models in edge computing environments. However, there are many drawbacks to the existing Simulators as few of them are tailored to satisfy particular requirements and do not meet all the criteria for this project.

Therefore, The paper "Souza et al. (2023)" introduces EdgeSimPy which addresses the shortcomings of existing simulators by providing a modular architecture that includes several functional abstractions for edge servers, network devices, and applications. It is completely developed in Python and is designed to model and evaluate resource management policies in edge-computing environments. EdgeSimPy allows detailed simulation scenarios by incorporating built-in models for user mobility, application composition, and power consumption, which also matches our requirements. The EdgeSimPy helps in the simulation of the algorithm in efficiently mapping the resources by addressing the requirements.

The paper uses EdgeSimPy to implement the ACO Algorithm and obtain the desired output. The ACO algorithm is tailored according to our requirements and the output is obtained after successful simulation. The dataset provided by the EdgeSimPy simulator is taken as the sample input and the simulation is performed by using them. This easy and convenient Python-based simulator helps in developing the algorithm and tailoring it to our needs to obtain the desired output.

3 Methodology

The objective of this research is to evaluate the performance and cost efficiency of auto-scaling dockerized container cloud resources using an advanced Ant Colony Optimization (ACO) algorithm. This evaluation is conducted by simulating scenarios within the Python-based cloud simulator, EdgeSimPy. The ACO algorithm is imported from the open-source Python library mealpy into the placement algorithm of EdgeSimPy Van Thieu and Mirjalili (2023). The imported ACO algorithm is tailored to the requirements and simulations are conducted accordingly. Additionally, the research will compare the ACO algorithm with traditional scaling algorithms such as First Come First Serve (FCFS) to demonstrate the effectiveness of the ACO algorithm in efficiently scaling cloud resources in response to user demands.

By analyzing various performance metrics and execution times, this study aims to provide a comprehensive assessment of the ACO algorithm's capability to enhance resource management in cloud environments. The findings highlight the potential improvements in scalability, load balancing, and overall system efficiency that can be achieved through the application of advanced ACO techniques.

3.1 EdgeSimPy Cloud Simulator

As the second step towards the implementation of the project, the Python-based Cloud simulator EdgeSimPy is used to develop a custom cloud resource scheduling scenario. After carefully analyzing different cloud simulators the EdgeSimPy was finally found suitable for the project because of three main reasons. Firstly the simulator consists

of six servers and each server consists of containers that can replicate the application provisioning method of Docker and can be easily integrated with the DockerHub repository. Secondly, it is developed in Python and therefore any Python library can be easily called or integrated into the simulator. Finally, it can be easily customized based on the requirements, its user-friendly Python interface helps in developing and running the algorithms Souza et al. (2023). The ACO algorithm from the open-source library mealpy can be efficiently customized into the placement algorithm of the cloud simulator as both the library and simulator are Python-based therefore minimizing the time in configuring the algorithm in different languages or simulators.

The proposed scenario includes six servers and a set of different services and each service should be assigned to its suitable servers based on the services. The architecture diagram of EdgeSimPy is shown in the figure 2 The EdgeSimPy simulator has two sets of datasets that can be used in this scenario and from that, 'sample_dataset2' is used for the simulation. All the services are predefined in the dataset. The objective of the simulation is for the algorithm to schedule smaller services to the smallest free resource efficiently and bigger services should be scheduled to the biggest free resources. Similarly, the FCFS algorithm was also tested in the EdgeSimPy Cloud Simulator to compare and contrast the results obtained.

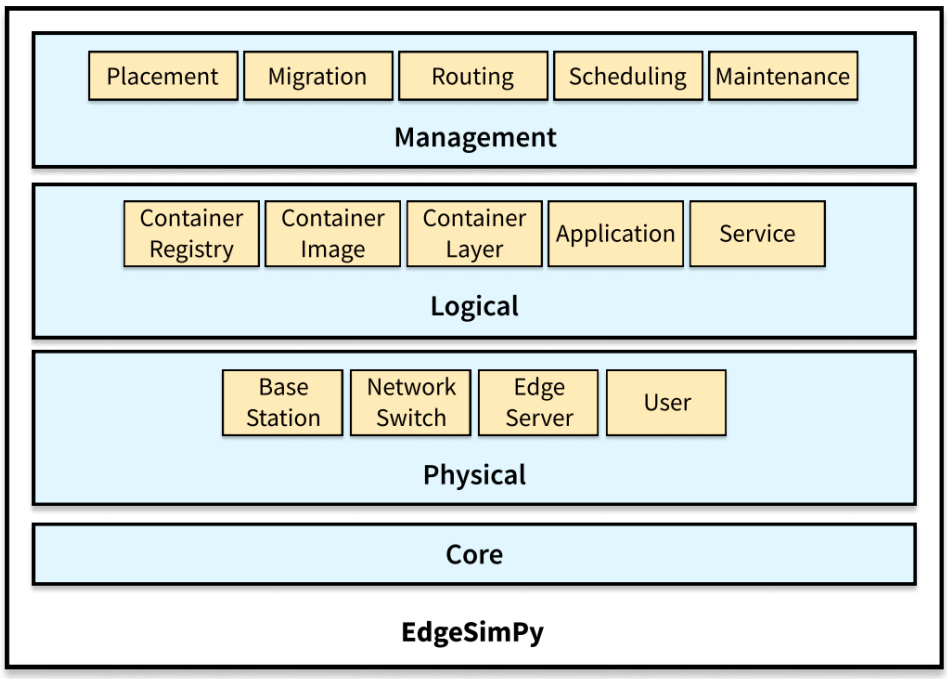

Figure 2: EdgeSimPy Architecture Souza et al. (2023)

For the simulation, the server capacities and service characteristics were defined in the EdgeSimPy dataset. The ACO algorithm was customized to match these scheduling requirements, ensuring services were allocated to the most suitable resources. Key parameters of the algorithm, including pheromone levels were adjusted to optimize performance. Similarly, the traditional FCFS algorithm was also tailored to this scenario and run to compare FCFS and ACO algorithms.

Data collection involved multiple simulation iterations to gather performance metrics

such as service completion time, resource utilization, and operational costs. Measurements focused on tracking service completion times and load distribution across servers.

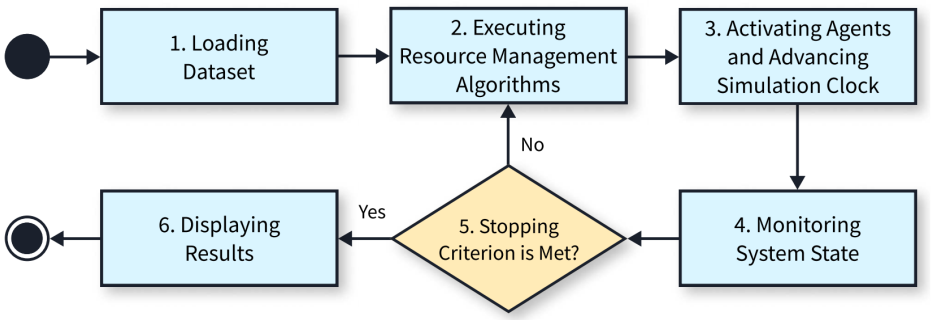

Figure 3: EdgeSimPy's Simulation Workflow Souza et al. (2023)

3.2 Proposed Scheduling Algorithm: ACO

The fundamental principle of Ant Colony Optimization is to stimulate and follow the foraging behavior of the ant colonies. The ant group's specialty is that whenever it searches for food, it leaves a chemical known as pheromone to communicate with each other. At first, the ants start searching for food in random ways, and when one of them finds a path to the food source they leave the chemical pheromone on the path. Therefore once an ant finds food and leaves its pheromone other ants will follow this trail by sensing the pheromone on the ground. Soon all the ants start coming to the food source thus finding the smallest way as there are huge amounts of pheromones on the ground to the food sourceDorigo et al. (2006). Therefore, it is clear that the ACO algorithm can be used and applied to any integrative problem as it will efficiently find the smallest path required to execute a job.

In ACO algorithms there is an evaporation rate, which refers to the process of pheromone trials which was led by the ants decreasing and vanishing over time if no new pheromones are laid by the antsEbadinezhad (2020). This means that the older the path becomes the more it loses its ability to give optimal solution. Therefore, in the proposed ACO algorithm a new method is implemented. To make sure continuous optimization is followed instead of pheromone evaporation, the algorithm uses parameters like pheromone importance, intent factor, and exploration factor to control and distribute the solutions. Therefore the proposed algorithm performs without worrying about the evaporation rate.

Here in this paper, the ACO algorithm is used by calling the library mealpy Van Thieu and Mirjalili (2023). It is an open-source library for meta-heuristic algorithms in Python. The ACO algorithm seamlessly finds the best path to the solution that is it will allocate the resources to the exact servers based on the parameters described. After calling the ACO algorithm from the mealpy library and tailoring it with the placement algorithm in the EdgeSimPy simulator based on the project requirement the ACO algorithm will efficiently allocate the exact resources for the oncoming services thus seamlessly distributing the loads at a low cost of operation. The algorithm gives the execution time, service-to-server allocation, best solution, and pheromone importance to calculate the performance of our proposed algorithm Zhao and Stankovic (1989).

4 Design Specification

This research focuses on developing an effective cloud resource scheduling solution by using advanced ACO algorithms and frameworks. The main components for the project are cloud simulator EdgeSimPy, to run the simulation, placement algorithm ACO from the mealpy library, and parameters that need to be defined to achieve the optimal resource allocation in a simulated cloud environment.

For the smooth working of this project, high-end resources are required. To minimize the cost and expense that will be caused by these resources a Cloud simulator EdgeSimPy is used to simulate the same cloud scenario. The simulator is run using a free platform for Python code execution called Google Colab which is connected to the Google Compute Engine and provides the necessary RAM and Disk space to run the simulation.

The ACO algorithm is customized and additional features are added to cover the main objective of the paper which is to scale resources cost-efficiently to servers by minimizing the total execution time and to distribute resources to servers based on their size and availability. To calculate the variance in resource utilization across edge servers and to estimate the execution time of services an '**objective_function(solution)**' is defined. This function initializes arrays to track the total load on each server, initializes sets to track which servers are used, and initializes a list to store execution times for services. The service assignment logic is also implemented inside this function. This function enables the code to iterate through the solution array and assign services to servers based on the optimized solution obtained from the algorithm.

To check if the target server has enough capacity to handle the service's CPU demand a capacity check logic is also developed in the code. A logic to check the least execution time and total execution time is also implemented in the code. It estimates the execution time based on the service's CPU demand and the server's CPU capacity. Additionally, a logic to calculate the variance in resource utilization is added to the code and a logic is implemented to add a penalty if not all servers are utilized. A function to retrieve the number of services assigned to the servers is also initialized in the code. The figure 4 shows the flowchart of the proposed ACO algorithm.

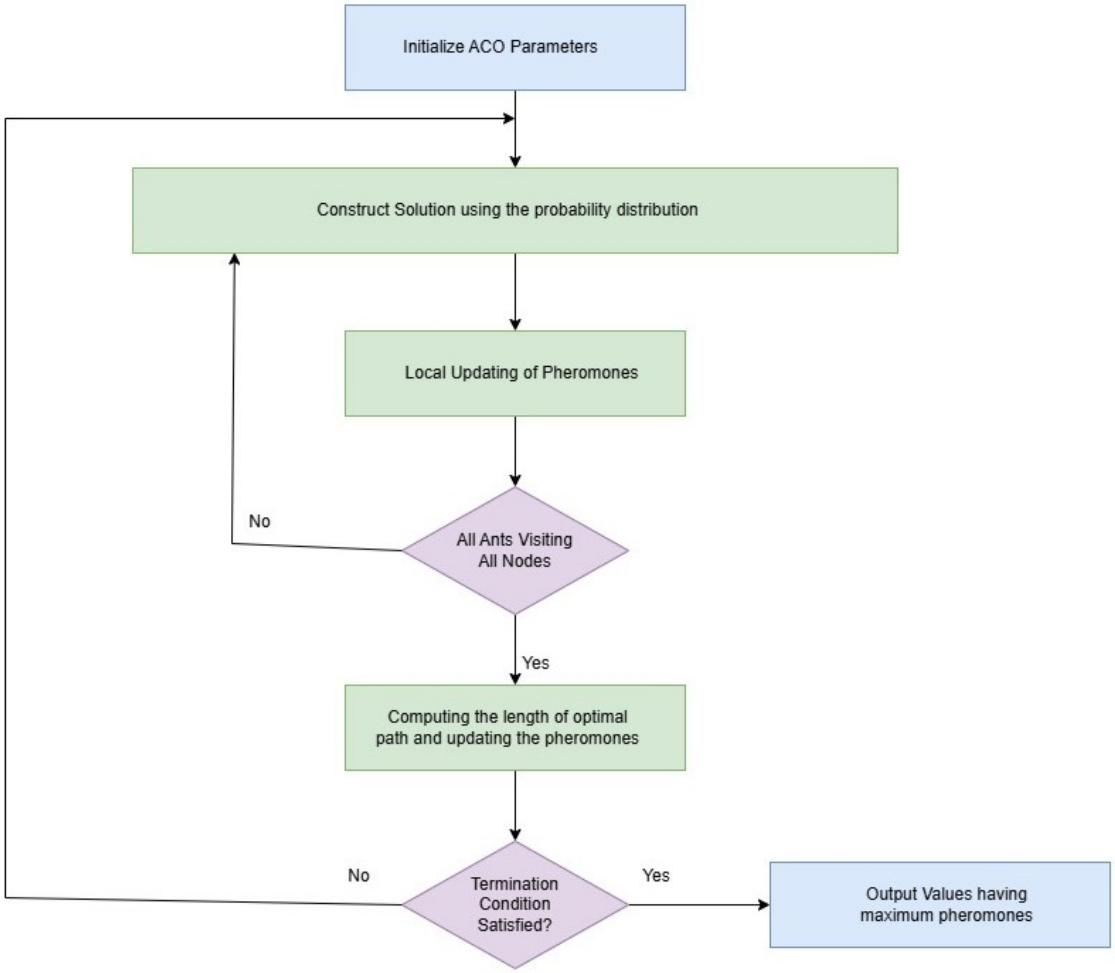

Figure 4: ACO Flowchart

5 Implementation

For the implementation of this project several factors needed to be structured and organized to obtain the desired results.

5.1 Creating EdgeSimPy Workspace

For the simulation, the EdgeSimPy Simulator environment was established by connecting it to the Google Compute Engine via Google Colab. This setup helped in facilitating the necessary computational power required for the cloud resource scheduling experiment. The requirements for the simulators were installed to satisfy the dependency issues. The ACO algorithm was then integrated within the placement algorithm framework by importing the mealpy library. The ACO algorithm is then developed to track the servers assigned to each service. This Python library for algorithms enabled the efficient application of ACO to our specific use case.

5.2 Integrating the proposed ACO Algorithm

The '**my_algorithm**' function is defined within this framework to enclose the entire optimization process. The '**objective_function**' helps in calculating the load on each server and the corresponding variances in these loads. This variance is an important metric as it shows how effectively the services are distributed across the servers. A penalty is added to the variance in case all the available servers are not used. This ensures that all the available servers are used for resource allocation thus ensuring maximum utilization.

To structure the optimization problem the '**problem_dict**' dictionary is defined. This dictionary includes the '**bounds**' for the variables, the objective function, and the optimization directions. The bounds initiate the possibility of allocation of each server to each service, ensuring that each service will only be assigned to a valid server index. Similarly, the '**obj_func**' in the dictionary is linked to the '**objective_function**', which helps in evaluating the quality of each solution.

Several parameters for the efficient functioning of ACO are described in the algorithm to showcase the resource scheduling such as:

epoch: This parameter describes and sets the number of iterations the algorithm will run. That is each epoch is a representation of one complete cycle of the algorithm's operation.

pop_size: This parameter sets the number of solutions considered for each iteration in other words the number of ants to find the right path. The number of ants describes how fast they can find an efficient path.

sample_count: This parameter defines the amount of samples that are drawn in each iteration. This parameter has a great influence as it can determine the diversity and quality of solutions.

intent_factor: This parameter affects how strongly the pheromone trails influence the search process. It also has a great influence on the selection probability of solutions based on their quality.

zeta: The zeta parameter helps in controlling the influence of the pheromone trail on the solution construction. The strong pheromone influence leads to finding the best solutions.

These parameters are crucial as they balance the entire process within the optimization algorithm. These parameters also have an impact on the algorithm's ability to find high-quality solutions efficiently. Once the parameters were configured, the '**solve**' method was used to execute the algorithm. This method applies the optimization to the problem defined in the '**problem_dict**'. Then the execution time is measured to evaluate the efficiency of the algorithm. The pseudo-code for the proposed ACO algorithm is represented in algorithm1.

Algorithm 1 Proposed Ant Colony Optimization Algorithm

```
Define Function MY_ALGORITHM(parameters)
    Define Function OBJECTIVE_FUNCTION(solution)
        total_load ← zeros(size of EdgeServers)
        assigned_servers ← empty set
        execution_times ← empty list
for each service_idx, server_idx in enumerate(solution) do
            server_idx ← round(server_idx)
            if server_idx invalid: continue
            total_load[server_idx] ← total_load[server_idx] + service.cpu_demand
            add server_idx to assigned_servers
            if edge_server has 'cpu':
                execution_time ← service.cpu_demand / edge_server.cpu
                append execution_time to execution_times
end for
    variance ← variance(total_load)
    if not all servers used: variance ← variance + 10
    return variance + sum(execution_times)
model ← initialize ACO with parameters
g_best ← model.solve(problem_dict)
best_solution ← g_best.solution
least_execution_time ← ∞
for each service_idx, edge_server_idx in enumerate(best_solution) do
        edge_server_idx ← round(edge_server_idx)
        if edge_server_idx invalid: continue
    if service not provisioned then
        if edge_server can host service then
            provision service to edge_server
        end if
    end if
        if edge_server has 'cpu':
            execution_time ← service.cpu_demand / edge_server.cpu
            least_execution_time ← min(least_execution_time, execution_time)
end for
print results and summaries
```

The outputs from the cloud simulation provided a detailed understanding of the performance and behavior of the ACO algorithm. The result of the simulation explains the Epoch numbers in each provisioning, the number of provisioned services, the number of ants in each provisioning, the pheromone importance, the exploration factor zeta, the runtime of each epoch in each provisioning, the current best, global best, and the execution time. The obtained results of the simulation of the ACO algorithm's effectiveness in efficiently scheduling resources. The time taken by the algorithm to provision each service to a server helped in identifying the algorithm's impact. Similarly the outputs detailed which servers were assigned to different services thus providing a complete understanding of resource allocation dynamics.

5.3 Integrating the traditional FCFS Algorithm

To find out the efficiency of the ACO algorithm, the results obtained from the first simulation were compared with the results of the FCFS algorithm. The traditional FCFS algorithm allocated the resources to the servers on a first come first serve basis. The pseudo-code for traditional FCFS is represented in algorithm 2. Thus the comparison mainly focused on the execution time and resource provisioning efficiency of both algorithms. The results of FCFS and ACO highlighted the advantages of using ACO over FCFS in terms of optimizing cloud resource scheduling.

Algorithm 2 FCFS Service Placement Algorithm

```
total_load ← zeros(size of EdgeServers)
assigned_servers ← empty set
Measure start_time
for each service in Service.all() do
    server_idx ← 0 {Start with the first server}
    while server_idx ¡ length of EdgeServers do
        edge_server ← EdgeServer.all()[server_idx]
        if edge_server can host service then
            provision service to edge_server
            total_load[server_idx] ← total_load[server_idx] + service.cpu_demand
            add server_idx to assigned_servers
            break {Move to the next service}
        end if
        server_idx ← server_idx + 1
    end while
end for
Measure end_time
execution_time ← end_time - start_time
variance ← variance(total_load)
print results and summaries {Total load, variance, execution time, and provisioning status}
```

This research demonstrates that the advanced ACO algorithm significantly improves scalability, load balancing, and cost efficiency compared to traditional methods like FCFS. After conducting several simulations in the simulator with the proposed scenario the advanced ACO algorithm showed results that were much more efficient than the traditional FCFS algorithms. The scenario was tested with both algorithms to find and compare the efficiency, cost implications, and execution time.

6 Evaluation

The experiment was conducted on EdgeSimPy Cloud Simulator which was run on Google Colab. The platform Google Colab was connected to the Google Compute Engine to get the resources like RAM and Disk space required to run the simulation. Each server in the EdgeSimPy simulator has the properties of docker containers consisting of CPU, RAM, container image, disk, and layers. There are six servers in the EdgeSimPy simulator which is apt for the simulation. The ACO algorithm is integrated into the placement

algorithm of the simulator. The parameters which define the behavior and efficiency of the algorithm are defined. The dataset in the simulator is loaded to the algorithm and the simulation is executed.

The Cloud scenario has to be run multiple times with two different algorithms to compare and contrast the results. The first and second simulation was with the developed ACO algorithm to find out the efficiency of the proposed algorithm. The third simulation was with the traditional FCFS algorithm to find out the differences and similarities between these two algorithms.

6.1 Scenario 1 with 50 Ants

The first simulation uses the proposed ACO algorithm. The dataset was loaded in the algorithm. The parameters which determine the behavior and efficiency of the algorithm are defined as follows. The epoch value was set to 100 (which means there will be 100 iterations in search of the optimal solution), the pop_size value was set to 50 (total of 50 ants at each iteration), the sample_count value was set to 25 (the number of solutions sampled from the probability density function generated by the pheromones is 25), the intent_factor value was set to 0.5 (affects the influence of pheromone concentration), and the zeta value was set to 1.0 (maintain balance between convergence speed and solution diversity). The table 1 shows the output obtained.

Table 1: ACO Algorithm Output with 50 Ants

Number of Ants	Pheromones	Exploration Factor	Server	Best Solution (variance)	Least Execution Time (sec)
50	0.5	1.0	Server_1	0.45490336	0.08
50	0.5	1.0	Server_2	2.12304361	0.08
50	0.5	1.0	Server_3	3.84371329	0.08
50	0.5	1.0	Server_4	2.55933762	0.08
50	0.5	1.0	Server_5	0.50089009	0.08
50	0.5	1.0	Server_6	4.82672705	0.08

The optimal service placement is indicated by the solution vector [0.45490336 2.12304361 3.84371329 2.55933762 0.50089009 4.82672705] . This vector is the representation of services assigned to specific servers which are the dockerized containers. The current best value is the best solution found in the current epoch and is obtained 10.958333333333334 for the first 7 iterations and the global best which represents the best solution found across all epochs is obtained 10.958333333333334 till 7th iteration. This means that the algorithm has not yet found a better solution which is the shortest path. The average runtime for these epoch's were around 0.017 to 0.018 seconds. The figure 5 shows that.

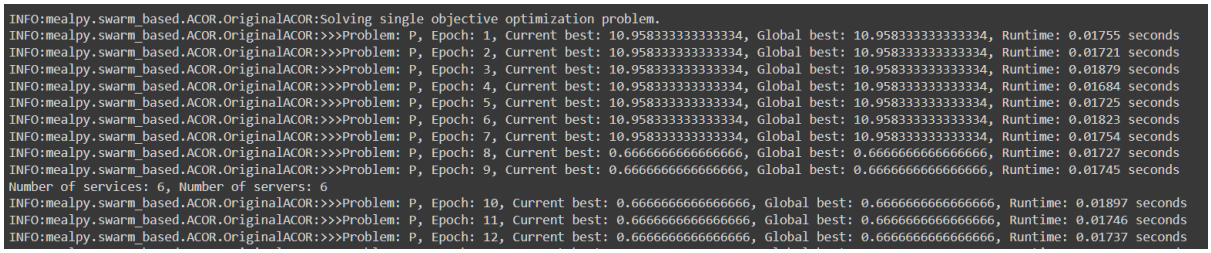

Figure 5: Output of First 12 epoch's

However, from 8th iteration the current best value obtained is 0.6666666666666666 which means that the algorithm discovers a significantly better solution and this new value becomes the new global best value. Therefore, from epoch 8 onwards, the values of current best and global best remains at 0.6666666666666666 which means that this solution is stable. The current best and global best remains at 0.6666666666666666 till the end of the provisioning indicating that the algorithm has converged to an optimal solution. The average runtime for these epoch's were around 0.017 to 0.025 seconds. The figure 6 shows that.

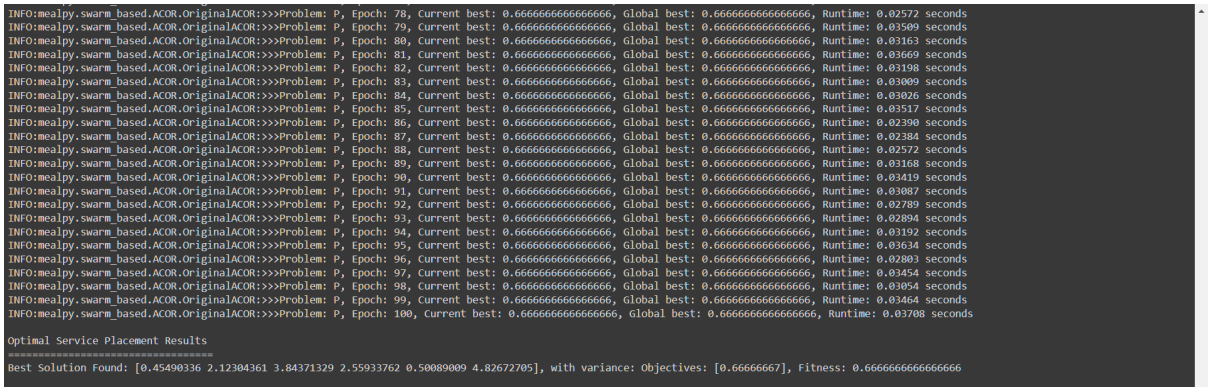

Figure 6: Output of rest of the epoch's and best solution found with 50 Ants

The number of provisioned services is 6 out of 6 which means that all the services were efficiently scheduled to the servers. The list of services that were assigned to each server is obtained. The service_1 is assigned to server_2, service_2 is assigned to server_3, service_3 is assigned to server_4, service_4 is assigned to server_6, service_5 is assigned to server_5, service_6 is assigned to server_1 which shows that the resource is allocated to different servers based on the size and the availability of the servers. The runtime of each iteration is available and the execution time obtained is 0.08 seconds and least execution time is 0.08 seconds which means that it used less time to find the smallest path and provision the resources. The figure 7 shows that.

Figure 7: Output of Service to Server Provisioning with 50 Ants

6.2 Scenario 2 with 25 Ants

The second simulation uses the ACO algorithm with parameters set to a minimum. The epoch value was set to 50, the pop_size value was set to 25 (25 ants at each iteration), the sample_count value was set to 15 (the number of solutions sampled from the probability density function generated by the pheromones is 25), the intent_factor value was set to 0.5 (affects the influence of pheromone concentration), and the zeta value was set to 1.0. The table 2 shows the output obtained.

Table 2: ACO Algorithm Output with 25 Ants

Number of Ants	Pheromones	Exploration Factor	Server	Best Solution (variance)	Least Execution Time (sec)
25	0.5	1.0	Server_1	5.	0.08
25	0.5	1.0	Server_2	1.02972597	0.08
25	0.5	1.0	Server_3	0.	0.08
25	0.5	1.0	Server_4	2.90972066	0.08
25	0.5	1.0	Server_5	4.12903057	0.08
25	0.5	1.0	Server_6	1.88101963	0.08

In the second simulation, the optimal service placement is indicated by the solution vector[5. 1.02972597 0. 2.90972066 4.12903057 1.88101963]. The current best and global best value obtained is 11.0 for the first 3 iterations. This means that the algorithm has not yet found a better solution or the shortest path. The average runtime for these epoch's were around 0.010 to 0.011 seconds. In epoch 4, a slightly better solution than the previous epochs are found which is indicated by the current best and global best values at 10.958333333333334. The runtime increases slightly to 0.01251 seconds. The figure 8 represents that.

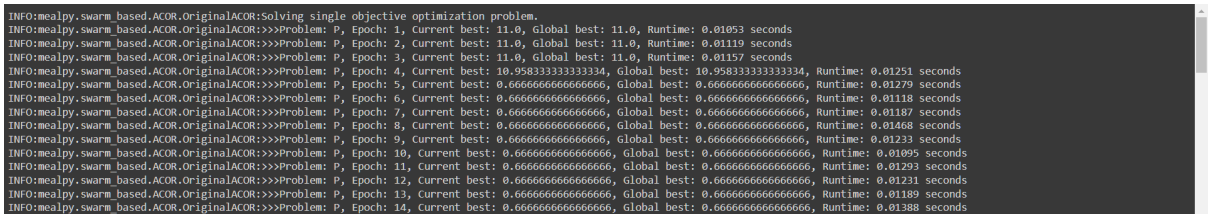

Figure 8: Output of First 10 Epoch's

However, from epoch 5 the algorithm finds the optimal solution which is indicated by the current and global best values of 0.6666666666666666. The average runtime remains stable at 0.01279 seconds, showing that the algorithm efficiently identified this perfect solution without a significant increase in computational cost. The figure 9 represents that.

Similarly in the second simulation, all the resources are efficiently scheduled. The service_1 is assigned to server_3, service_2 is assigned to server_1, service_3 is assigned to server_6, service_4 is assigned to server_4, service_5 is assigned to server_5, service_6 is assigned to server_2 which shows that the resource is allocated to different servers based on the size and the availability of the servers with the help of the optimal solution. The

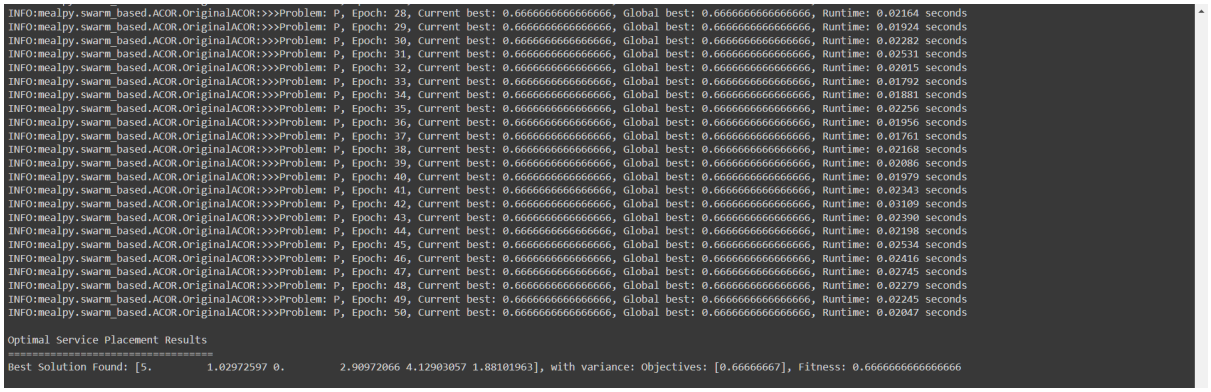

Figure 9: Output of rest of the Epoch's and best solution found with 25 Ants

runtime of each iteration is available and the execution time obtained is 0.12 seconds and least execution time is 0.08 seconds which means that it used less time to find the smallest path and provision the resources. The figure 10 represents that.

Figure 10: Output of Service to Server Provisioning with 25 Ants

6.3 Scenario 3 with FCFS Algorithm

This simulation uses the traditional FCFS algorithm for scheduling. The purpose of FCFS algorithm is to schedule the first requests to the first available server. This traditional algorithm do not check the service size or resource size and directly assigns the services to the first server. For this simulation there are 6 servers and the dataset with services is loaded in the algorithm developed. After running the simulation the table 3 is obtained.

Table 3: FCFS Algorithm Output

Load on Server	Service	Server	Variance of Server Load	Execution Time (sec)
6	Service_1	Server_1	5	0.0
0	Service_2	Server_1	5	0.0
0	Service_3	Server_1	5	0.0
0	Service_4	Server_1	5	0.0
0	Service_5	Server_1	5	0.0
0	Service_6	Server_1	5	0.0

The total load on Servers for each provisioning is different. The FCFS algorithm does not split the services and schedule it to the next server. The load on the server is obtained to be 6 for the first provisioning indicating that all the services are assigned to the server_1 in the first provisioning. That means FCFS assigns all services to the nearest server till it gets completely occupied rather than distributing the services based on the service size. The figure12 represents that. The variance of server load obtained

is 5 which means that all the loads were placed on a single Server which reflects the imbalance in resource distribution. The execution time obtained is 0.0 that is because the FCFS algorithm is not searching for the optimal path but just assigning the services to the first server thus making it execute faster which is not ideal for a real situation.

Figure 11: Optmial Placement results of FCFS

Figure 12: Output of Service to Server Provisioning with FCFS

6.4 Discussion

After conducting three simulations, two with the proposed ACO algorithm and one with the traditional FCFS algorithm it is observed that our proposed design achieved the desired output. It efficiently scheduled resources by finding the optimal solution with less time thus achieving cost efficiency. In both the ACO simulation, the ACO algorithm demonstrated a balanced load distribution with minimal execution times. To site, in the first scenario with 50 Ants, the algorithm found the optimal solution by the 8th iteration and in the second scenario with 25 ants, the algorithm found the optimal solution by the 5th iteration. Whereas, the FCFS algorithm assigned all the services to the first available server without checking the resource availability or load distribution. The variance in the server load showed that this algorithm is inefficient for efficient resource scheduling and therefore could not be used for distributing resources efficiently. This leads us to the fact that the proposed ACO algorithm is a milestone in container resource scheduling problems.

However, as far as the maximum performance of the given ACO algorithm is concerned, there are certain aspects, which could be modified to gain enhanced results. The fact that the algorithm depends on parameters as many of ants and the influence of a pheromone, one can suppose that it performance may greatly depend on the chosen parameters. This sensitivity to parameter settings might result in different suboptimal outcomes in different scenarios especially in the settings whereby workload volatility and available resources vary greatly. In the future to improve the algorithm it may be relevant to look into ways to make these parameters dynamic and self-adjusting in response to the situation, possibly using concepts related to machine learning.

7 Conclusion and Future Work

The goal of this study was to determine if better resource scheduling at less cost in the cloud could be done with the use of Docker containers and enhanced auto-scaling ACO algorithm. The initial aim included determining the advantages of the ACO algorithm in order to define the logical utilization of limited resources and analyze the working compared with the traditional FCFS algorithm. The simulations proved that the proposed ACO algorithm improvises the FCFS algorithm in giving efficient utilization of resources, lesser execution times of a job, and better performances of the system. In both the situations where the application of the ACO algorithm was made, it was able to find efficacious solutions in a very few number of iterations, as in the current case it was with 50 ants' iteration 8 and with 25 ants, iteration 5. This led to proper load sharing among the various servers hence exemplifying how the algorithm comes in handy when different tasks are being run in cloud environments.

FCFS on the other hand was incapable of colorizing the jobs according to the availability of resources or distribution of load across several servers all the jobs got assigned to one server which clearly led to inefficiencies and imbalances. The above results showed the effectiveness of the proposed ACO algorithm in solving the challenges associated with containerized resource scheduling in a cloud environment.

The future work of this project could focus on developing a dynamic and adaptive version of the ACO algorithm which adjusts the parameters in real-time based on the current workload and resource availability. Similarly, this research can focus on developing an energy-efficient model by integrating energy-aware metrics into the algorithm. In this way, this work opens the possibility for future research that can extend these directions, likely to provide stronger, more efficient as well as more readily scalable methods of scheduling resources in cloud environments for commercial applications.

References

Ahmed, A. and Sabyasachi, A. S. (2014). Cloud computing simulators: A detailed survey and future direction, *2014 IEEE international advance computing conference (IACC)*, IEEE, pp. 866–872.

Dorigo, M. (2007). Ant colony optimization, *Scholarpedia* **2**(3): 1461.

Dorigo, M., Birattari, M. and Stutzle, T. (2006). Ant colony optimization, *IEEE computational intelligence magazine* **1**(4): 28–39.

Dorigo, M., Maniezzo, V. and Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents, *IEEE transactions on systems, man, and cybernetics, part b (cybernetics)* **26**(1): 29–41.

Dorigo, M. and Stützle, T. (2019). *Ant colony optimization: overview and recent advances*, Springer.

Ebadinezhad, S. (2020). Deaco: Adopting dynamic evaporation strategy to enhance aco algorithm for the traveling salesman problem, *Engineering Applications of Artificial Intelligence* **92**: 103649.

Goyal, S. (2010). A survey on travelling salesman problem, *Midwest instruction and computing symposium*, pp. 1–9.

Li, K., Xu, G., Zhao, G., Dong, Y. and Wang, D. (2011). Cloud task scheduling based on load balancing ant colony optimization, *2011 sixth annual ChinaGrid conference*, IEEE, pp. 3–9.

Mireslami, S., Rakai, L., Wang, M. and Far, B. H. (2015). Minimizing deployment cost of cloud-based web application with guaranteed qos, *2015 IEEE Global Communications Conference (GLOBECOM)*, IEEE, pp. 1–6.

Mishra, R. and Jaiswal, A. (2012). Ant colony optimization: A solution of load balancing in cloud, *International Journal of Web & Semantic Technology* **3**(2): 33.

Sharma, N., Garg, P. et al. (2022). Ant colony based optimization model for qos-based task scheduling in cloud computing environment, *Measurement: Sensors* **24**: 100531.

Souza, P. S., Ferreto, T. and Calheiros, R. N. (2023). Edgesimpy: Python-based modeling and simulation of edge computing resource management policies, *Future Generation Computer Systems* **148**: 446–459.

Srirama, S. N., Adhikari, M. and Paul, S. (2020). Application deployment using containers with auto-scaling for microservices in cloud environment, *Journal of Network and Computer Applications* **160**: 102629.

Stützle, T., López-Ibáñez, M. and Dorigo, M. (2011). A concise overview of applications of ant colony optimization, *Wiley encyclopedia of operations research and management science* **2**: 896–911.

Tawfeek, M. A., El-Sisi, A., Keshk, A. E. and Torkey, F. A. (2013). Cloud task scheduling based on ant colony optimization, *2013 8th international conference on computer engineering & systems (ICCES)*, IEEE, pp. 64–69.

Van Thieu, N. and Mirjalili, S. (2023). Mealpy: An open-source library for latest meta-heuristic algorithms in python, *Journal of Systems Architecture* **139**: 102871.

Wan, X., Guan, X., Wang, T., Bai, G. and Choi, B.-Y. (2018). Application deployment using microservice and docker containers: Framework and optimization, *Journal of Network and Computer Applications* **119**: 97–109.

Zhao, W. and Stankovic, J. A. (1989). Performance analysis of fcfs and improved fcfs scheduling algorithms for dynamic real-time computer systems, *1989 Real-Time Systems Symposium*, IEEE Computer Society, pp. 156–157.

Student Name: Kurian George

Student ID: X22191437

MSCCLOUD1_A

MSc Research Project

Supervisor: Punit Gupta

Optimizing Resource Scheduling in Cloud Environments with Docker Containers and Advanced Auto-Scaling Algorithms

1. How does your work compare with other published techniques?

My work proposed a new method of cloud resource scheduling by integrating Docker containers with my developed version of Ant Colony Optimization (ACO) algorithm. This method proved to be cost effective as the runtime is lower when compared the same scenario with the published First Come First Serve (FCFS) algorithm. The published techniques like First Come First Serve (FCFS), Round Robin (RR), Genetic Algorithm (GA), and Particle Swarm Optimization (PSO) all have several drawbacks such as overloading of virtual machines, suboptimal task handling, and have increased runtime which make them unsuitable for efficient resource scheduling purposes. My developed ACO algorithm overcomes all these issues faced by traditional methods by dynamically scheduling services based on service size and server availability, thereby minimizing runtime and reducing cloud utilization costs.

2. Why have you considered only FCFS and ACO?

I have considered FCFS and ACO for this study because these two algorithms represent two contrasting approaches to cloud resource scheduling. The FCFS is a traditional algorithm and ACO is an advanced scheduling algorithm. It serves as a good baseline for comparison in this study. The FCFS is commonly used in various cloud environments because of its simplicity and ease of implementation. However, there are some limitations to this model like lack of dynamic optimization, allocating services based on arrival time and not based on service size. This often leads to inefficient resource utilization in cloud environments.

Whereas the advanced algorithm like ACO is designed to optimize resource scheduling by dynamically searching for optimal solutions in real time thus reducing cloud utilization costs. But certain factors like evaporation rate need to be calculated and adjusted. Therefore, I chose to modify the ACO algorithm to satisfy the specific requirements of my cloud scheduling scenario, ensuring that it not only optimizes performance but also addresses the limitations found in traditional methods like FCFS.

3. Are all the decimal digits in your evaluation results significant?

The values of solution vector, current best, global best, and least execution time are having decimal values. All these values are important as they represent the results. But in cloud resource scheduling, a difference in the third or fourth decimal place might not have a substantial impact on overall performance or cost. In a practical application, only the first few decimal digits are typically significant for decision making. However, the entire decimal digits are used in the results to maintain accuracy in comparisons and to maintain consistency throughout the evaluation process. The comparison between scenario one, two and three can be effectively clarified with the help of these decimal digits. This detail in decimal digits will also help in carrying out any future work related to this study.