Stage 3: New scheduling algorithms – Design Document

1. **Project Title:** Autonomous job scheduler

GitHub: https://github.com/SnakeCN21/COMP3100-Group-Project/tree/S3_Mark/Stage3

2. **Student Details**

Student ID	Student Name
45141916	Jiahui Lin

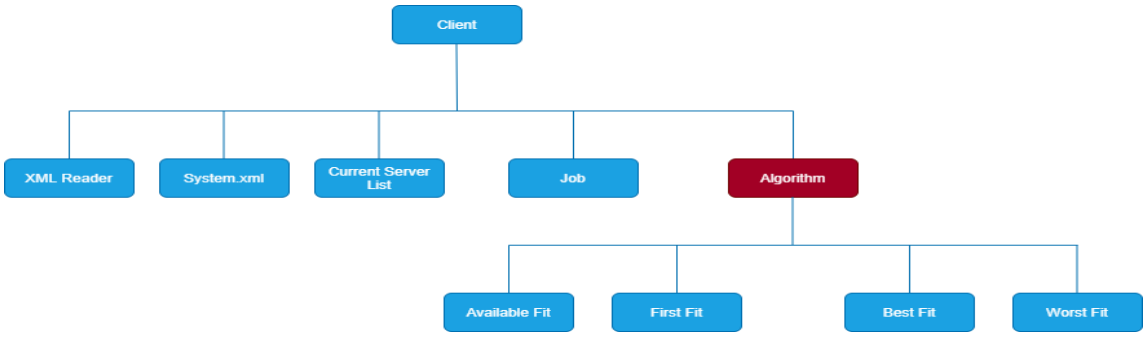
3. **Introduction**

This stage allows students to design and perform a new scheduling algorithm that optimizes one or more performance metrics in terms of job scheduling. A comprehensive comparison would be made to all relevant algorithms (Available Fit, First Fit, Best Fit, and Worst Fit) to comparing their performance metrics such as resource utilization, execution time, turnaround time, and total cost. As a result, a champion algorithm would be selected if it optimizes the targeted performance metrics.

4. **Problem definition/formulation**

Time efficiency is significantly important in this day, as we can see the agile development methodologies are commonly used in most of the organizations for the fast pace delivery of products. Job scheduling in the distributed system is not an exception, well use of resources and time enable us to complete more tasks in less time, based on this idea, my goal is to design and implement an algorithm that minimizes the turnaround time and waiting time. Rather than buying new devices or upgrading the existing ones, implement a good algorithm can save the unnecessary expense on the hardware, and have long term benefits on business growth.

5. **Design Consideration and Preliminaries:**



Assumptions:

- There must a be server with initial resources which capable of running all different kinds of jobs.
- A server can run multiple jobs at once if it does not exceed the resources capacity.

- Every time the RESC message is issued, server will always return the updated server status.

XML Reader Class: Extracts and read all the server types with their initial resource capacity from the System.xml. This information is stored inside of the XML reader class and can be referred when an XML reader object is created.

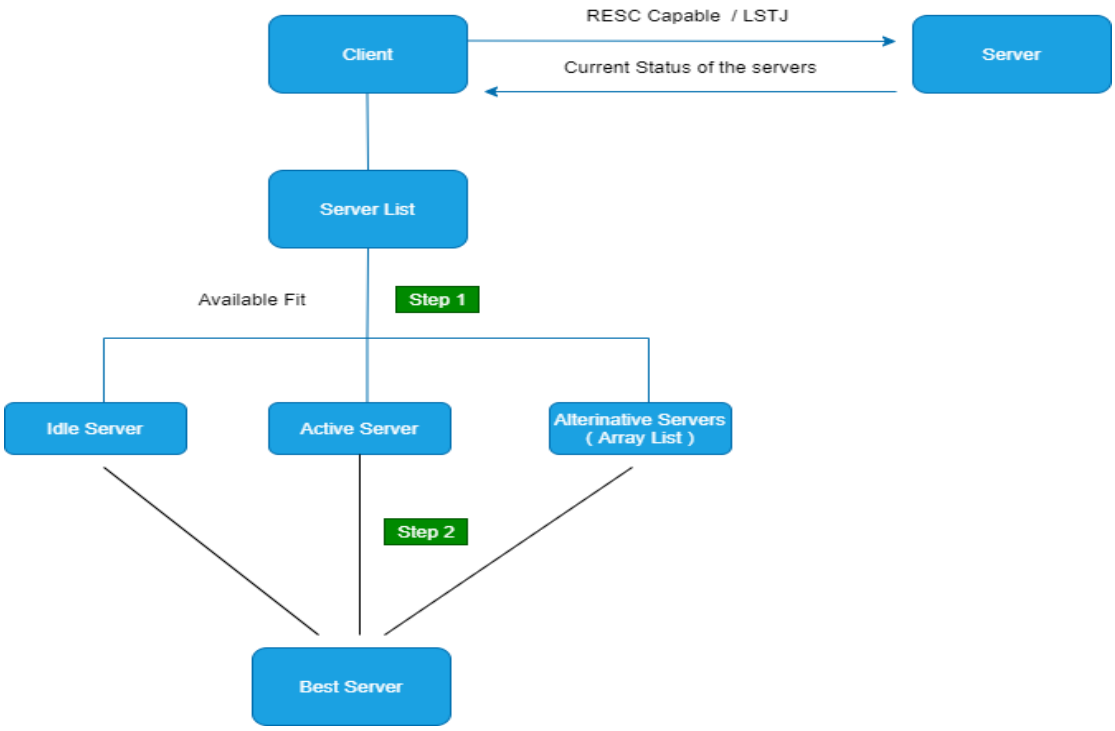
Jobs (Array): An array created to store each job information.

Current Server List (Array List): When a **RESC Capable** command with job information is issued and sent from the client side , the server **(ds-server)** will return current status of the servers that are initially capable to schedule such job. This information is fetched by the client and storing into an Array List. When an algorithm is performed, the client will loop over this list to find a targeted server.

Algorithm interface: The client simulator is capable of performing a specific algorithm one at time, however for this stage only one algorithm has been implemented which is called Available fit, and it called by inserting the argument **(" -a af")** along with the command to run client simulator at the terminal. And if not, argument is inserted, the client simulator will run AllToLarget algorithm (from stage 1) as the default.

6. Algorithm Description - Available Fit

The aim of this algorithm is to improve the turnaround time by waiting time before the assigning a new job to a specific server. Our customized algorithm can be implemented in two steps, **step 1** is to sort the servers into three different groups and **step 2** is to find a best server within these three groups.



6.1 Algorithm – Step 1 - Sorting

In **Step 1,** when **RESC Capable** command issued, the client will fetch the status of initially capable servers returned by the **ds-server.** Categorize the servers into three different groups **idle server, active server, and alternative servers. Active Server –** server is currently available and have enough resource to schedule such job.

Idle Servers – server is currently inactive but has the minimum booting up time.

Alternative severs – Servers that are active and was initially fit for the job, but current has not enough resources to schedule such job.

Initialize all the variables

```
private static HashMap<String, String> availableFit(ArrayList<HashMap<String, String>> serverList,Stag
    // sort the server into three different groups
    HashMap < String, String > bestServer = new HashMap < String, String > ();
    HashMap < String, String > activeServer = new HashMap < String, String > ();
    HashMap < String, String > idleServer = new HashMap < String, String > ();
    ArrayList<HashMap<String, String>> alternativeServers = new ArrayList<HashMap<String, String>>();
```

Idle Server = server object
Active Server = server object
Alternative Servers = ArrayList of server objects.
Best Server = server object, our targeted server, would be return by the function.

Loop through the server List and sort them into different groups

```
for (int i = 0; i< serverList.size();i++) {
    int serverCoreSize = Integer.parseInt(serverList.get(i).get(CPU_CORES));
    int serverDisk = Integer.parseInt(serverList.get(i).get(DISK_SPACE));
    int serverMemory = Integer.parseInt(serverList.get(i).get(MEMORY));
    int serverState = Integer.parseInt(serverList.get(i).get(SERVER_STATE));
    int tempTime = Integer.parseInt(serverList.get(i).get(AVAILABLE_TIME));

  // looping through the list trying to find idle server with least waiting time
    if(serverState !=3 && isFitted(serverList.get(i),cpuCores,memory,disk)&& (tempTime< bootingTime ||(tempTime== bootingTime
        idleServer = serverList.get(i);
        bootingTime=tempTime;
    }else if(serverState ==3  &&activeServer.get(MEMORY)==null&&isFitted(serverList.get(i),cpuCores,memory,disk)) {
      activeServer = serverList.get(i);
    }else if(!isFitted(serverList.get(i),cpuCores,memory,disk)) {
      alternativeServers.add(serverList.get(i));
    }
  }
```

For Server in a ServerList

1.Obtain the server status

2.If the Server's stage != 3 and it currently has the enough resources to schedule the job and has the least booting up time, assign it to the variable **idleServer.**

3. If the **first found** server is the list with the server state == 3 and it currently has the enough resources to schedule the job, assign it to the variable called **activeServer.**

4. If the server is initially fit but currently does not have enough resource to schedule the job, add it to the Array List called **alternativeServers**.

Note: Every time a **RESC Capable** command is issued, client would fetch a new server list.

If either idle server is found or an active server is found, the second step of the algorithm will not be executed. And if both idle server and active server are found, the algorithm will assign the **active server** as the **best server**, because the active server is already up on running whereas the idle server still needs time for booting up, thus our algorithm will choose the server that is **first available**.

6.2 Algorithm – Step 2 - Get least waiting server

This step is performed if both idle server and active are not found. When sorting the server list in the previous step, we have sorted the active servers with insufficient current resource to the array list called **AlternativeServers.** A server with least execution time would be returned for this method. Our goal is to find soonest available server because a job might not be able to be schedule until one or more currently waiting or running jobs of the server have been completed. A server with least execution time indicates that this server is going free up its resources soonest among the alternative server list.

```
private static HashMap<String, String> getLeastWaitingServer(Stage2FF client,ArrayList<HashMap<String, String>>
  //HashMap < String, String > serverMap = new HashMap < String, String > ();
    int minExecutionTime = Integer.MAX_VALUE;
    HashMap<String, String> leastWaitingServer = new HashMap<String, String>();

    for(int i = 0; i< serverList.size();i++) {
        int temp = 0;
        client.sendMsg("LSTJ "+serverList.get(i).get(SERVER_TYPE)+" "+serverList.get(i).get(SERVER_ID));
        String msg = client.getMsg();
            while(!msg.equals(DOT)){

                client.sendMsg(OK);
                msg = client.getMsg();
                temp += getExecutionTime(msg);
            }
            if(temp!=0 && temp<minExecutionTime) {
                minExecutionTime = temp;
                leaseWaitingServer = serverList.get(i);
            }
    }
    return leastWaitingServer;
}
```

LeastWaitingServer = server with **minimum** total execution time of the jobs.
For server in alternative servers:

1. Fetch the job list of a server by issuing **LSTJ msg** with server type and server id.

For jobs in a server

2. Gain and add up the execution time of each of a server

Return and assign the **best server** to the server with least total execution time.

6.3 Additional methods

1. isFitted()

```
private static boolean isFitted(HashMap<String, String> server,int cpuCore,int memory, int disk) {
    int serverCpu = Integer.parseInt(server.get(CPU_CORES));
    int serverMemory = Integer.parseInt(server.get(MEMORY));
    int serverDisk = Integer.parseInt(server.get(DISK_SPACE));

    if(serverCpu>=cpuCore && serverMemory >= memory && serverDisk >=disk ) {
        return true;
    }else {
        return false;
    }
}
```

Simple Boolean assertion method, checks give a given server has the resource capacity to schedule a job.

2. getExecutionTime()

```
private static int getExecutionTime(String serverMsg) {
    HashMap < String, String > serverMap = new HashMap < String, String > ();
    int executionTime = Integer.MAX_VALUE;
    String line[] = serverMsg.split(SPLIT);
    if(line.length == 7){
        String jobState = line[1];
        String estimatedRunTime = line[3];
        executionTime = Integer.parseInt(estimatedRunTime);
    }
    return executionTime;
}
```

Method that extract job state and execution time from the given msg.

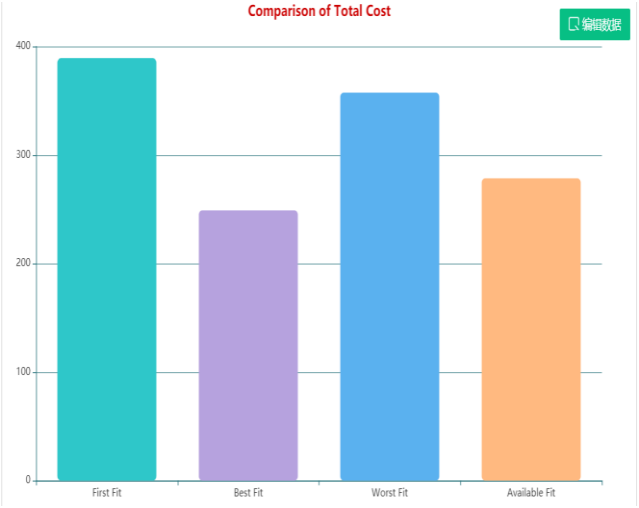
7. Algorithm Evaluation

The aim of this section is to test how effective does 'Available Fit' algorithm improves metrics in comparison to the other baseline algorithms (First Fit, Best Fit and Worst Fit). Ds-client simulator would be used to run those three baseline algorithms, and use my own client simulator to run 'Available Fit' algorithm.

There are five configs files chosen for demonstration, each has a different amount of workload and server types.

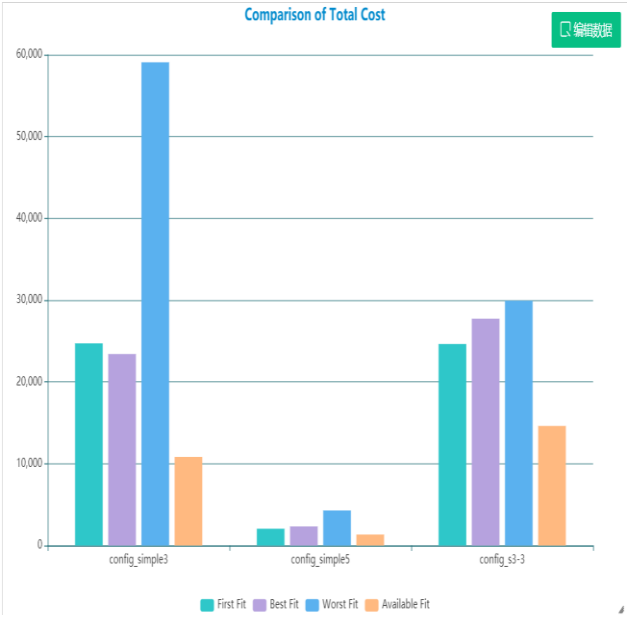
Config Files	Number of jobs	Workload
Config_simple1	50	light
Config_simple3	355	medium
Config_simple5	1000	heavy
ds-config-s3-3	1000	heavy

7.1 Comparison of Total Costs



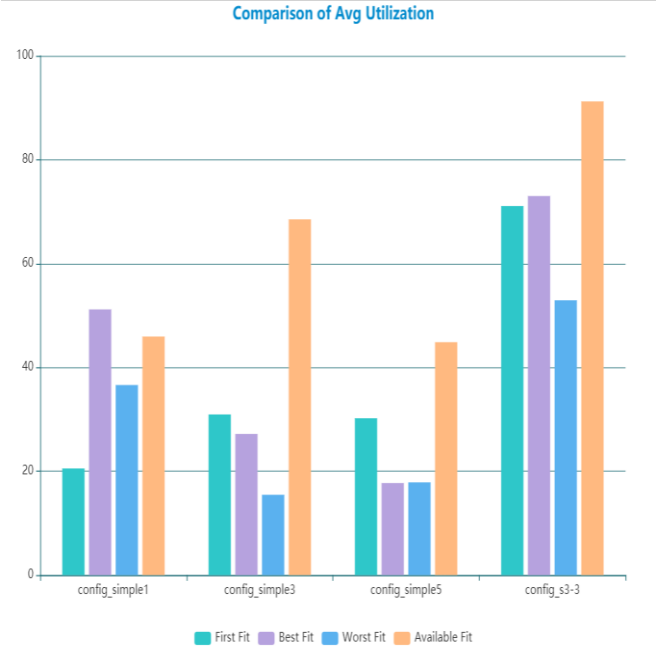
The diagram shows the performance of each algorithm for **config_simple1** and comparing to the best fit, available fit **does not improve total cost** under a **light workload environment**.

Note: It's been separate from the graph below because of the value range.



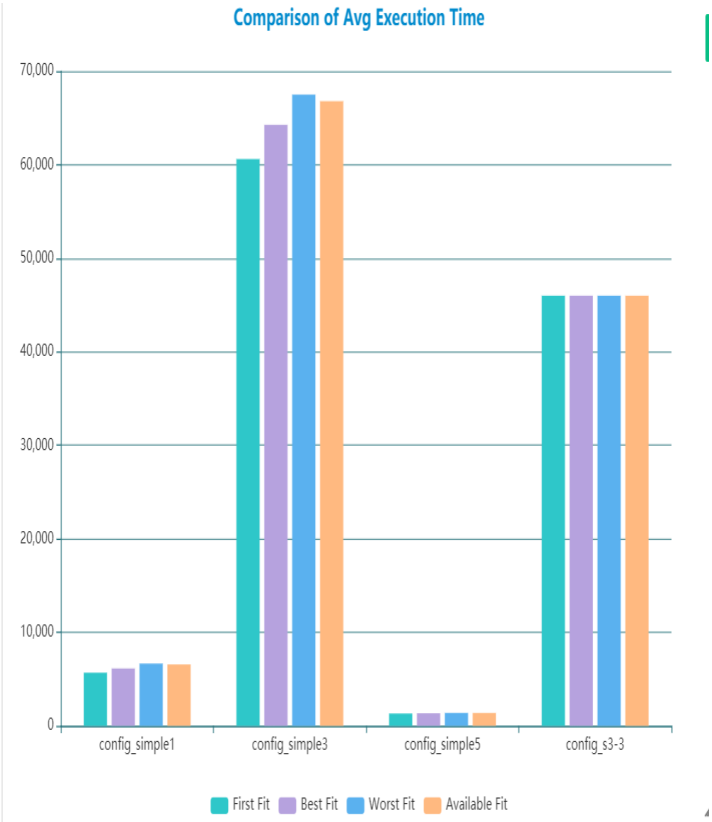
The diagram on the left shows the performance of each algorithm for **config_simple3, config_simple5 and config-s3-3.** As the result shown, available fit performs **exceptionally well** under **medium** and **heavy workload environments**.

7.2 Comparison of Avg utilization rate



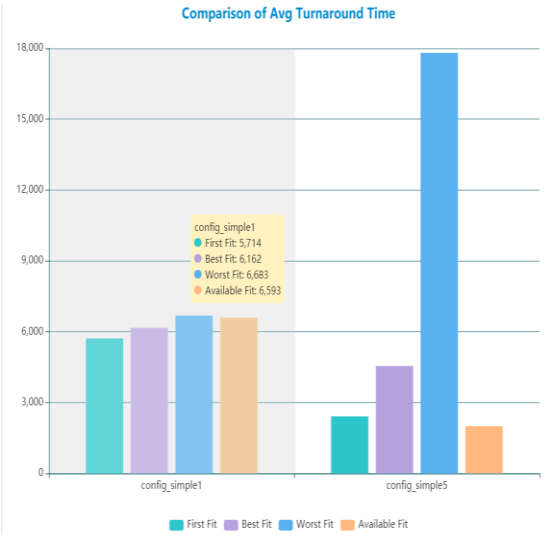
The diagram on the left shows the performance of the average utilization rate of each algorithm. As the result shown available fit **does not** improve the **average utilization rate** under all kinds of environments.

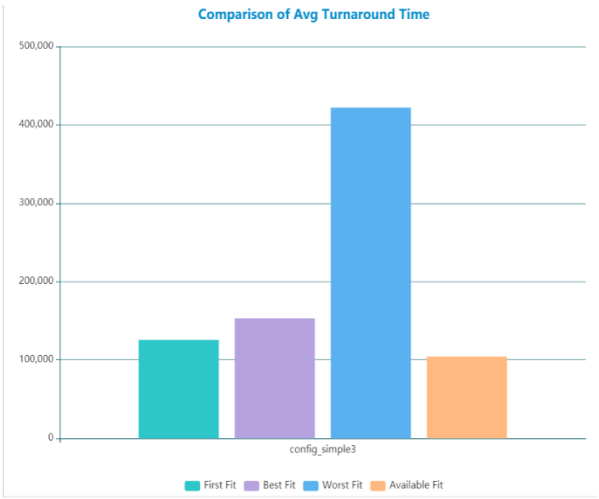
7.3 Comparison of Avg Execution Time

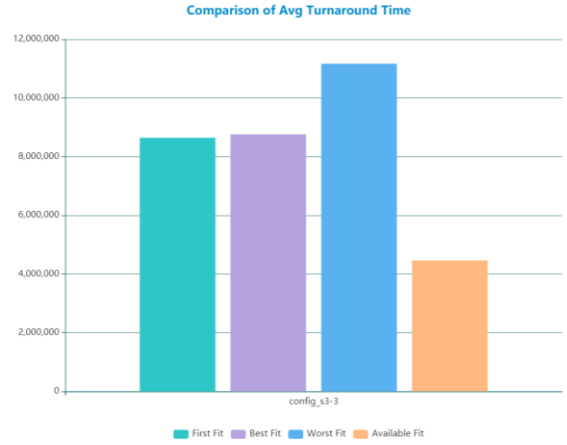


The diagram on the left shows the performance of **average execution** time of each algorithm. As the result shown, available fit does not improve avg execution time under all environments.

7.4 Comparison of Turnaround Time

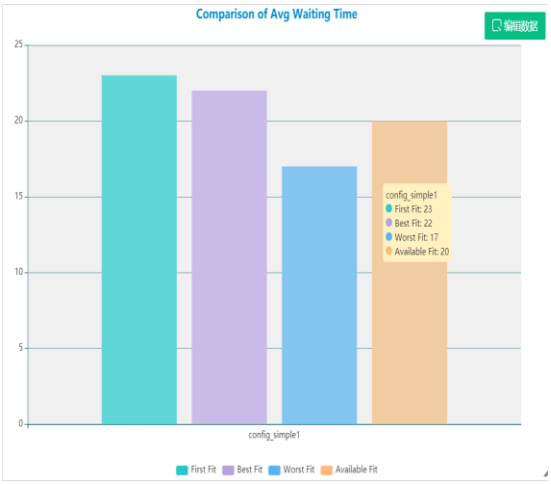




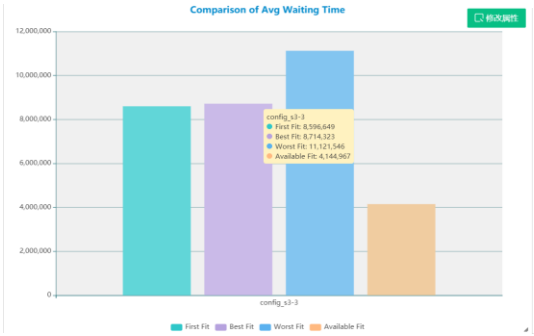


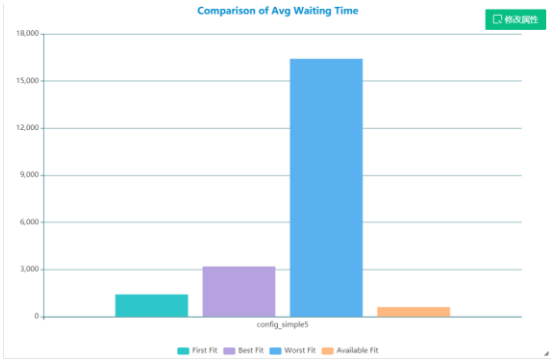
The diagrams above and left show the performance of the **turnaround time** of each algorithm for different config files. Again, there is **not improvement** under the **light workload environment**, but it does **perform well under medium and heavy workload environments**.

7.5 Comparison of Avg Waiting Time



The diagrams on the left show the performance of the avg waiting time of each algorithm under light workload environment, there is **no indication** that available fit can improve the avg waiting time under a **light workload environment.**





The diagrams on the left and above shows the performance of **avg waiting time** of each algorithm under **medium and heavy workload environment.** As the result shown, available fit performs **exceptionally well** under **medium** and **heavy workload environment**.

7.6 Summary Table

Config file	Metrics	First Fit	Best Fit	Worst Fit	**Available Fit**	**Improvement**
config_simple 0	Total Cost	$389.26	$249	$357.49	**$278.54**	✗
	Avg Utilization	20.54%	51.17%	36.61%	**45.95%**	✗
	Execution Time	5691	6140	6666	**6572**	✗
	Turnaround Time	5714	6162	6683	**6593**	✗
	Avg Waiting Time	23	22	17	**20**	✗
config_simple 3	Total Cost	$24,723	$23,416	$59,083.64	**$10,834.84**	✓
	Avg Utilization	30.94%	27.19%	15.47%	**68.52%**	✗
	Execution Time	60609	64273	67503	**66796**	✗
	Turnaround Time	125502	152919	422062	**104190**	✓
	Avg Waiting Time	64893	88646	354559	**37393**	✓
config_simple 5	Total Cost	$2,057.34	$2,340.77	$4,289.39	**$1,347.41**	✓
	Avg Utilization	30.21%	17.73%	17.86%	**44.86%**	✗
	Execution Time	1325	1353	1390	**1388**	✗
	Turnaround Time	2714	4547	17809	**1997**	✓
	Avg Waiting Time	1415	3194	16419	**608**	✓
config_s3-3	Total Cost	$27,643.11	$27,738.16	$29,933.08	**$14,624.27**	✓
	Avg Utilization	71.08%	73.01%	52.93%	**91.23%**	✗
	Execution Time	45997	45997	45997	**45997**	✗
	Turnaround Time	8642647	8760043	11167544	**4460964**	✓
	Avg Waiting Time	8596649	8714323	11121546	**44144967**	✓

8. Conclusion:

After sets of comparison, I found that my customized algorithm does not behave well under a light workload environment (less job / less timeframe) and it performs unsatisfactorily in terms of average execution time and avg utilization rate. However, under the medium and heavy workload environment, there are a significant improvement in total costs, turnaround time, and average waiting time. Overall, this algorithm does meet my expectation in terms of

reducing turnaround time and avg waiting time, but there is more upside potential can be made to improve the performance under the light workload environment.