

PROJECT REPORT

Γιώργος Μαθιουδάκης

Csd4674

Project Overview: Resource Allocation System with Cloud Integration

The primary goal of this project was to design and implement a cloud-based resource allocation system to optimize the distribution of computational resources across multiple users or tasks. Each user submits subtasks, and these subtasks are mapped to available resources in a way that balances time and cost. The system ensures efficient allocation and coordination between the users and a cloud provider while allowing for the dynamic update and reevaluation of utilities.

System Architecture

The system architecture consists of the following key components:

1. Users (Tasks): Each user represents a task with multiple subtasks. Subtasks are mapped to computational resources. Users run Flask-based EC2 instances to send allocation vectors and receive updated matrices.
2. AWS SQS: Acts as the intermediary for communication between users and the cloud provider.
3. Lambda Functions:
 - Lambda 1 processes incoming allocation vectors and stores them in a DynamoDB table.
 - Lambda 2 retrieves data from DynamoDB, generates an allocation matrix, and sends it to the cloud provider.
4. Cloud Provider: The cloud provider, running as a Flask application on an EC2 instance, calculates the updated execution time and expense matrices based on the allocation matrix and sends these updates back to the users.

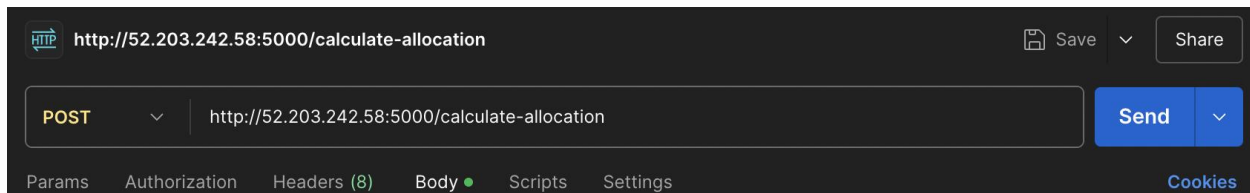
PHASE A - IMPLEMENTATION

Implementation Steps

- 1. Initial Task Submission by Users:**

- a. Users (S1, S2, S3) with subtasks (2, 3, and 4, respectively) submit their allocation vectors via Flask endpoints.

(Postman For Endpoints)



This is the output of user1 . Each user has the same process and similar output with different numbers obviously .

```
ec2-user@ip-172-31-86-79:~  
* Serving Flask app 'user_s1'  
* Debug mode: off  
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.  
* Running on all addresses (0.0.0.0)  
* Running on http://127.0.0.1:5000  
* Running on http://172.31.86.79:5000  
Press CTRL+C to quit  
User 1 - Best Allocation Vector: [0, 0, 1, 1, 0], Utility: 0.273972602739726  
Message sent to SQS: {'MD5OfMessageBody': 'a9ab512bd3ae41ba7750e1b448e76782', 'MessageId': 'b6bbc2a5-b853-4adc-9755-87224c5978a3', 'ResponseMetadata': {'RequestId': 'd95305fc-ecf6-5f6d-87dc-65f5522c5b21', 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': 'd95305fc-ecf6-5f6d-87dc-65f5522c5b21', 'date': 'Tue, 21 Jan 2025 15:26:18 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '106', 'connection': 'keep-alive'}, 'RetryAttempts': 0}}  
147.52.95.143 - - [21/Jan/2025 15:26:18] "POST /calculate-allocation HTTP/1.1" 200 -
```

- b. Allocation vectors represent the mapping of subtasks to resources.

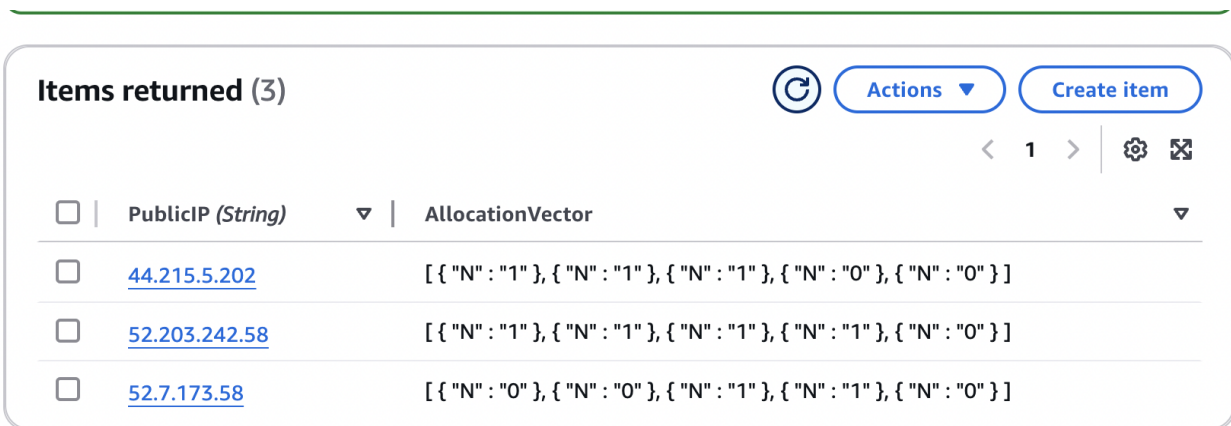
2. Communication via AWS SQS:



- a. The submitted allocation vectors, along with public IPs of the users, are sent to an SQS queue.

- b. SQS ensures asynchronous and reliable communication between users and the backend.

3. Lambda 1: Submission Handling:

- a. Lambda 1 is triggered by the SQS queue to process and store user submissions into a DynamoDB table.



Items returned (3)			Actions 	Create item
	PublicIP (String)	AllocationVector		
<input type="checkbox"/>	44.215.5.202	[{ "N" : "1" }, { "N" : "1" }, { "N" : "1" }, { "N" : "0" }, { "N" : "0" }]		
<input type="checkbox"/>	52.203.242.58	[{ "N" : "1" }, { "N" : "1" }, { "N" : "1" }, { "N" : "1" }, { "N" : "0" }]		
<input type="checkbox"/>	52.7.173.58	[{ "N" : "0" }, { "N" : "0" }, { "N" : "1" }, { "N" : "1" }, { "N" : "0" }]		

4. Lambda 2: Processing Submissions:

- a. Lambda 2 retrieves all submissions from DynamoDB once all users have submitted their vectors.
- b. It constructs the allocation matrix and forwards it to the cloud provider.

5. Cloud Provider: Matrix Updates:

- a. The cloud provider receives the public ips and the allocation vectors of which then , makes a matrix.

```
ec2-user@ip-172-31-27-160:~  
_/_m/'  
Last login: Sat Jan 18 16:15:06 2025 from 2.86.181.3  
[ec2-user@ip-172-31-27-160 ~]$ python3 provider.py  
* Serving Flask app 'provider'  
* Debug mode: off  
WARNING: This is a development server. Do not use it in a production deployment. Use a production W  
server instead.  
* Running on all addresses (0.0.0.0)  
* Running on http://127.0.0.1:5000  
* Running on http://172.31.27.160:5000  
Press CTRL+C to quit  
Received Allocation Matrix:  
[1, 1, 1, 0, 0]  
[1, 1, 1, 1, 0]  
[0, 0, 1, 1, 0]  
  
Received IPs:  
44.215.5.202  
52.203.242.58  
52.7.173.58
```

- b. The cloud provider calculates the updated **execution time (T)** and **expense (E)** matrices based on the allocation matrix.

```
Updated Execution Time Matrix:  
[0, 0, 12, 7.0, 0]  
[10, 8.4, 10.8, 0, 0]  
[8, 7.0, 9.600000000000001, 5.6, 0]  
  
Expense Matrix:  
[0, 0, 6.0, 6.3, 0]  
[5.0, 5.04, 5.400000000000001, 0, 0]  
[4.0, 4.2, 4.800000000000001, 5.04, 0]
```

- c. Updates are computed by considering resource sharing between users and the associated costs.

- d. The matrices are sent back to the users for reevaluation of utilities.

```
Data sent to User 1 (52.7.173.58). Response: 200 - {"new_utility":0.0823045267489712,"status":"success", "updated_execution_time_row":[0,0,12,7.0,0], "updated_expense_row":[0,0,6.0,6.3,0]}

Data sent to User 2 (44.215.5.202). Response: 200 - {"new_utility":0.07621951219512195,"status":"success", "updated_execution_time_row":[10,8.4,10.8,0,0], "updated_expense_row":[5.0,5.04,5.400000000000001,0,0]}

Data sent to User 3 (52.203.242.58). Response: 200 - {"new_utility":0.0723589001447178,"status":"success", "updated_execution_time_row":[8,7.0,9.600000000000001,5.6,0], "updated_expense_row":[4.0,4.2,4.800000000000001,5.04,0]}

18.212.251.40 - - [21/Jan/2025 15:26:49] "POST /receive-matrix HTTP/1.1" 200 -
```

6. Users Receive Updated Matrices:

- a. Users update their local execution time and expense matrices based on the new data.
- b. They calculate new utility scores for their subtasks, reflecting the shared resource usage and associated costs.

```
Updated Execution Time Row: [0, 0, 12, 7.0, 0]
Updated Expense Row: [0, 0, 6.0, 6.3, 0]
New Utility for User 1: 0.0823045267489712
184.73.242.139 - - [21/Jan/2025 15:26:49] "POST /update-matrix HTTP/1.1" 200 -
```

Again the process is shown for user1 only because it is very similar amongst all three users .

PHASE B – PETRI NETS

Objective of the Petri Net

The Petri Net was designed to model the interactions and processes in the resource allocation system, emphasizing the relationships between tasks (users), subtasks, resources, and the cloud provider. It captures the states and transitions of the system, enabling a structured analysis of resource allocation and task completion without explicitly addressing optimization.

Components of the Petri Net

1. Places (States):

- a. Represent different stages of task and resource management.
- b. Examples:
 - i. Tasks Submitted: Represents tasks submitted by users, awaiting resource allocation.
 - ii. Resources Allocated: Represents subtasks that are allocated to specific resources.
 - iii. Cloud Provider Processing: Represents the cloud provider calculating updated matrices.

- iv. Tasks Completed: Represents tasks successfully completed.

2. Transitions:

- a. Represent the events or actions causing the system to move from one state to another.
- b. Examples:
 - i. Submit Task: Transition from Tasks Submitted to Resources Allocated.
 - ii. Allocate Resources: Transition to distribute subtasks to resources.
 - iii. Process in Cloud Provider: Transition to update execution time and expense matrices.
 - iv. Receive Updated Matrices: Transition when users receive updates from the cloud provider.

3. Tokens:

- a. Represent the "presence" of tasks, subtasks, or resources in a specific state.
- b. Movement of tokens through transitions simulates the progression of tasks.

Key States in the Petri Net

1. Initial State:

- a. All tokens are in the Tasks Submitted place.
- b. Represents the starting point where all users have submitted their tasks but no allocation has been performed.

2. Intermediate State:

- a. Tokens are distributed between Resources Allocated, Cloud Provider Processing, and pending allocation states.
- b. Reflects the ongoing resource-sharing and cloud processing stages.

3. Final State:

- a. All tokens are in the Tasks Completed place.
- b. Represents successful allocation, processing, and completion of all tasks.

Reachability Graph

The reachability graph complements the Petri Net by showing:

- **All Possible States:** Each node in the graph represents a possible distribution of tokens among places.
- **Transitions:** Edges between nodes represent transitions triggered by events (e.g., task submission, resource allocation).
- **System Behavior:** Provides a complete view of how the system progresses from the initial state to the final state.

The graph illustrates:

- How tasks move from submission to completion.

- Dependencies between tasks, resources, and the cloud provider.
- Scenarios where resource contention or delays might occur.

Modeling Steps

1. Task Submission:

- a. Tokens start in Tasks Submitted, representing users submitting their allocation vectors.
- b. A transition (Submit Task) moves tokens to Resources Allocated.

2. Resource Allocation:

- a. Tokens are distributed among resources based on the allocation matrix.
- b. The Allocate Resources transition assigns resources to subtasks.

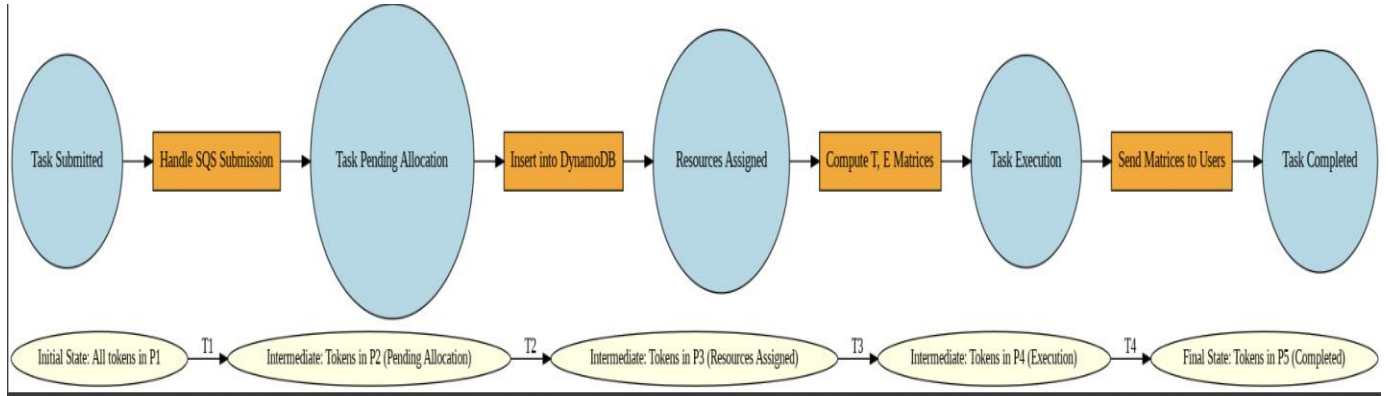
3. Cloud Provider Processing:

- a. Tokens move to Cloud Provider Processing when the cloud provider calculates updated execution time and expense matrices.
- b. The Process in Cloud Provider transition updates matrices and sends them back to users.

4. Task Completion:

- a. Tokens move to Tasks Completed as users receive updated matrices, calculate new utilities, and complete their tasks.

b. The Receive Updated Matrices transition represents users finishing their calculations.



In summary, this project successfully implemented a comprehensive resource allocation system that demonstrated effective interaction between users, cloud providers, and computational resources. Through dynamic resource sharing, updated task utilities, and real-time communication, the system met the intended goals of fairness, efficiency, and scalability.