Project: Simulation of a pair of Tandem Queues using Event Stack

Wahid Uz Zaman (wzz5219@psu.edu)

Project Description:

Implementation:

In this project, I implemented an `EventStack` (a doubly linked list) in `eventstack.py` and then used it to simulate a network of `TandemQueues` in `tandemqueuesimulator.py`. The `TandemQueueSimulator` is designed to handle any number of queues in a tandem network. To set up the simulation, the user provides a Poisson arrival rate for the first queue and exponential service rates for each queue. Any sequential tandem queue network (with two or more queues) should satisfy Jackson's theorem for performance metrics for each queue, assuming each queue follows an M/M/1 structure. For stability, the arrival rate must be smaller than the service rates.

To verify the tandem queue simulation (with two queues, each following M/M/1 and the first queue's departure time matching the second queue's arrival time), I checked Jackson's theorem metrics—mean sojourn time, mean number of jobs, and utilization—for each queue. I also validated Little's Law for each queue.

Additionally, I tested with a non-Poisson (uniform distribution between 3.4 and 5.5) arrival rate for the first queue and calculated the performance metrics.

In `main.py`, I detailed two experimental setups:

1. Arrival rate (Poisson) = 1.0, Service rates = [1.5, 1.2]
2. Arrival rate (Poisson) = 2.0, Service rates = [2.5, 2.2]

For each of these setups, I also ran the simulation with a non-Poisson arrival rate, keeping the service rates unchanged.

Generalization:

My current `TandemQueueSimulator` can already handle any number of tandem queues. For flexibility with arrival and service rates, I can add parameters allowing the user to specify any distribution for these rates across the queues.

To extend this to a general queuing network, like a Jackson network, I would add support for transition probabilities between queues. The user could provide a transition probability matrix, where each element defines the probability of moving from one

queue to another. With this setup, upon a queue departure, the job may either leave the network entirely (with a certain probability) or transition to another queue based on the probabilities in the matrix.

Alternatively, the user could provide a graph-like structure, where directed edges represent transitions between queues, and edge weights indicate the transition probabilities. In the departure event handling, I would adjust the code to determine the next queue based on the defined probability distribution.

Network Partition and Each partition has its own event stack:

In this setup, the network is divided into partitions, each with its own event stack. When events are processed independently within each partition, inter-partition dependencies can arise. This happens when an event in one partition must be processed before certain events in another partition, leading to potential out-of-order processing. This issue occurs because each stack operates independently, which can cause sequences where an event's prerequisites haven't been met.

To address this, there are two possible approaches:

1. **Synchronization**: Implement a method to synchronize events between stacks, ensuring events are processed in the correct order even when they originate from different stacks.
2. **Rollback**: If an out-of-sequence event is detected, the affected stack could roll back any events processed after the point in time where the out-of-sequence event was supposed to occur. This rollback would involve undoing the effects of those subsequent events, restoring the state to its original condition. The out-of-sequence event is then processed, followed by reprocessing the rolled-back events. This rollback scheme can be added on top of my code easily.

Simulation Results:

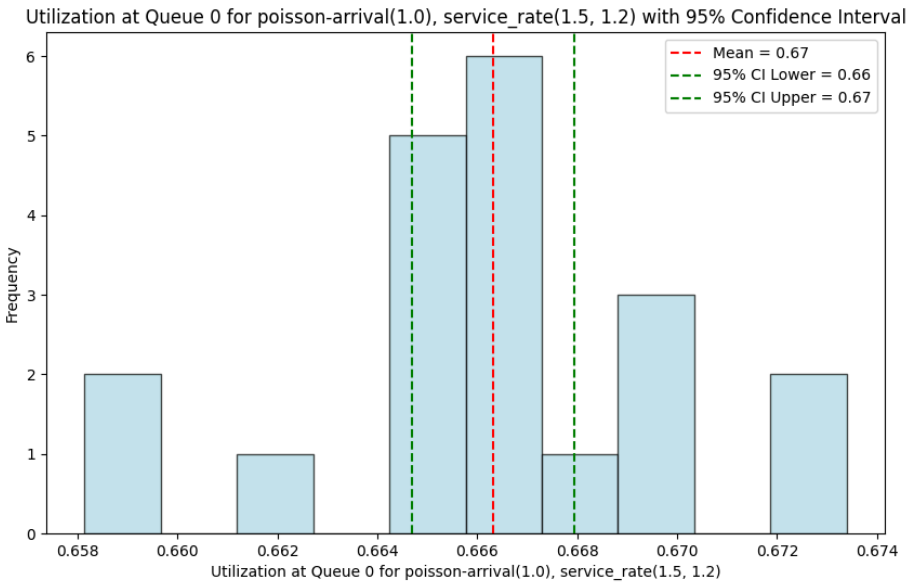
I created a `main.py` script to run various simulations with different configurations:

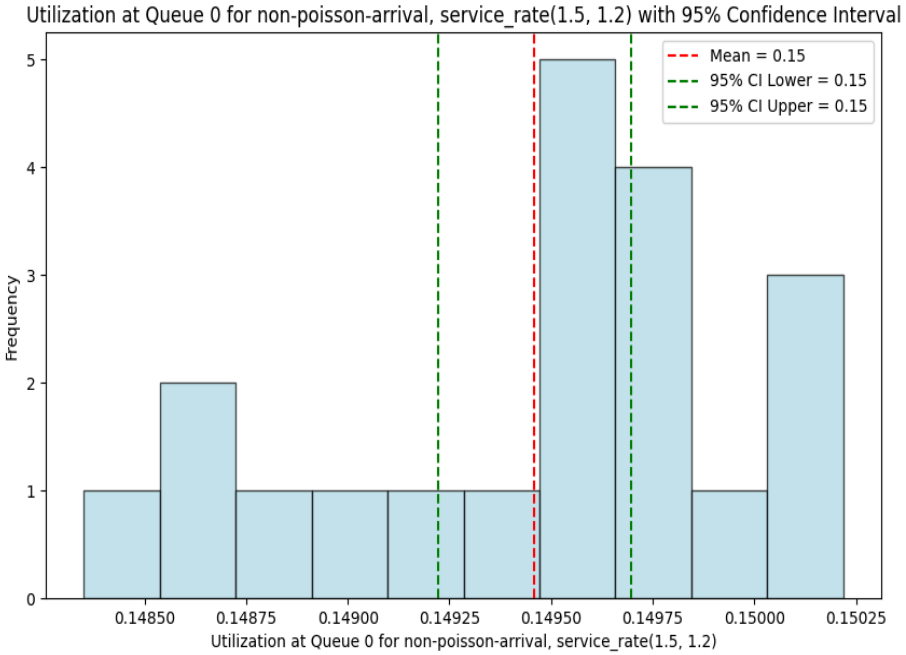
- Poisson arrival rate of 1.0 and service rates [1.5, 1.2]
- Non-Poisson arrival rate and service rates [1.5, 1.2]
- Poisson arrival rate of 2.0 and service rates [2.5, 2.2]
- Non-Poisson arrival rate and service rates [2.5, 2.2]

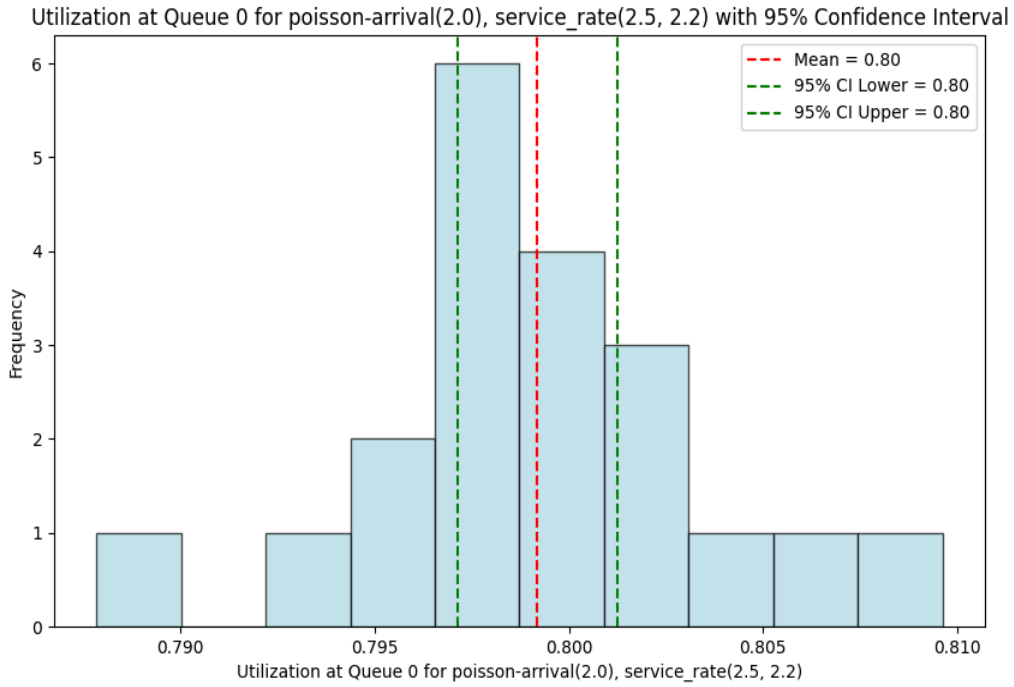
I evaluated three key metrics for these simulations. Below, I'll present four graphs (for queue 0 only) to illustrate each metric across the different runs listed above. However,

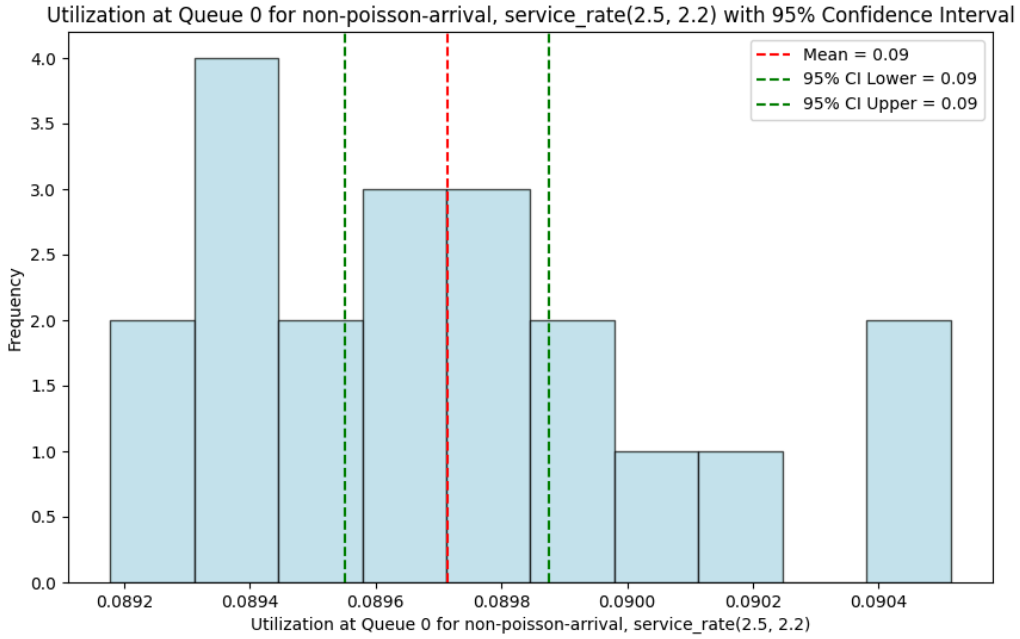
the Python script (`main.py`) will also generate corresponding graphs for queue 1. Also, my statistical result nearly matches Jackson's formulas for Poisson arrivals and exponential service rates.

Metric: Queue0 Utilization

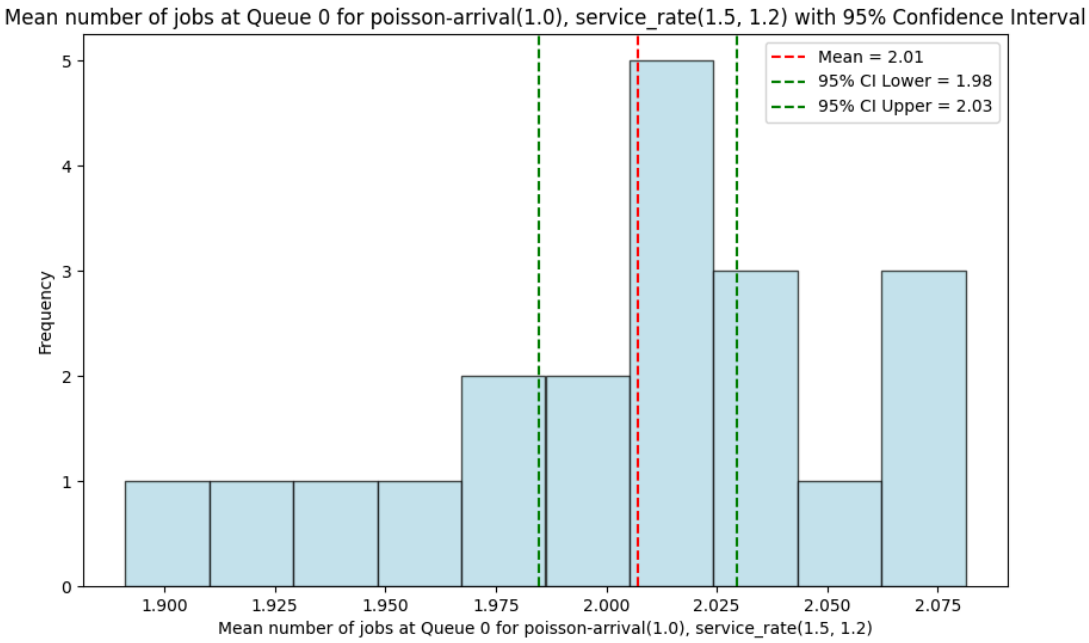


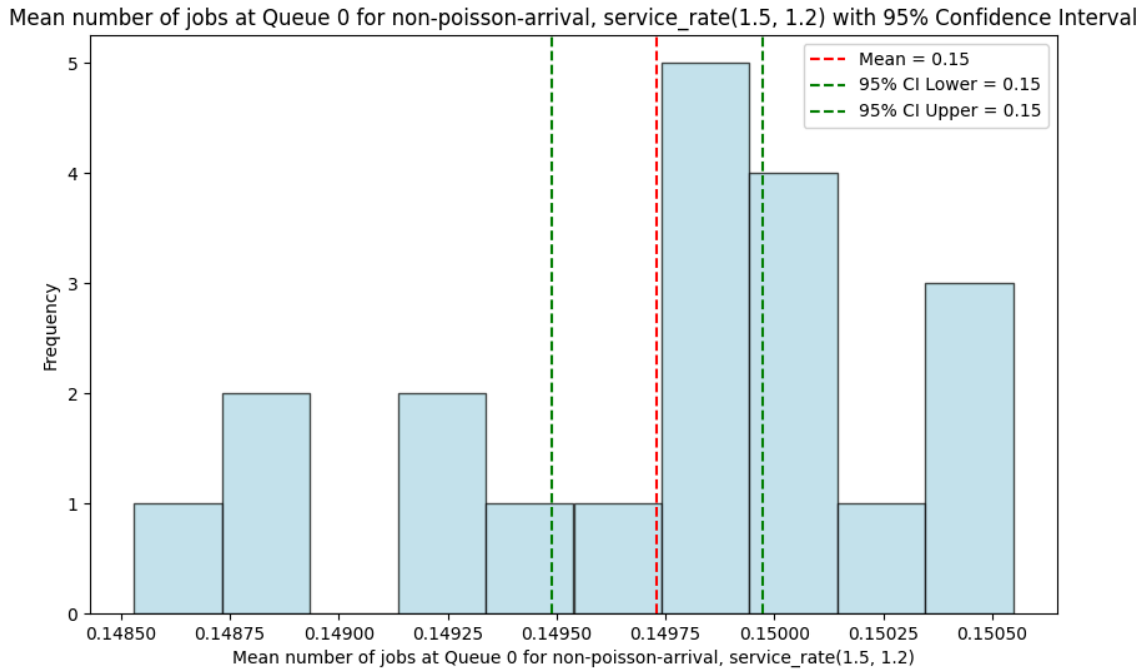


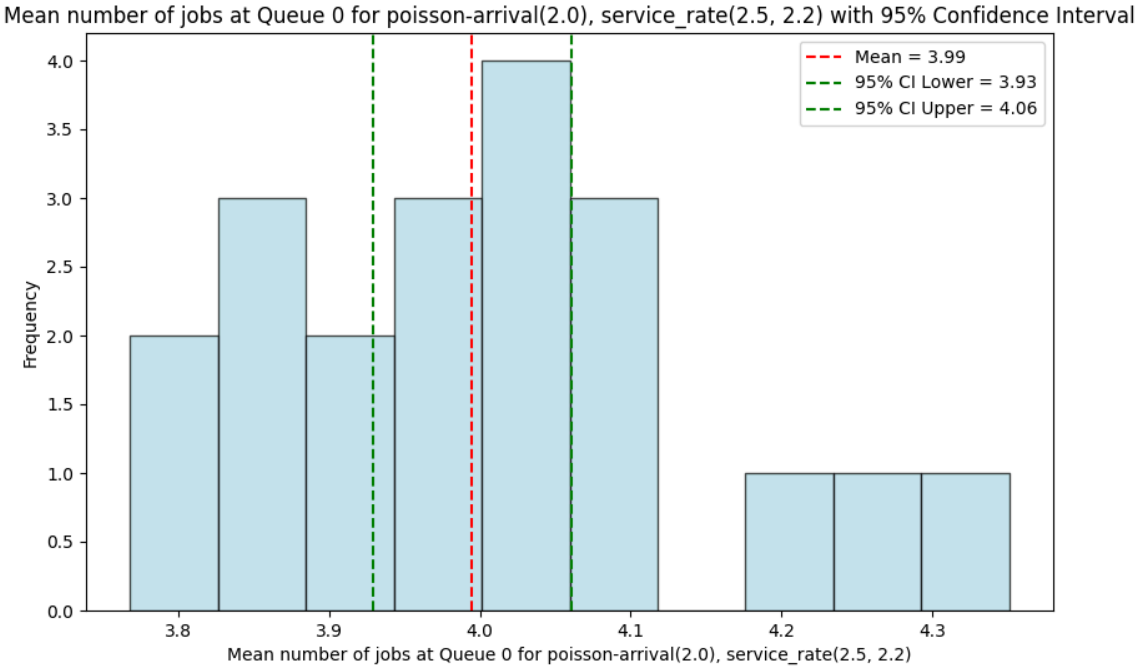

Utilization at Queue 0 for poisson-arrival(2.0), service_rate(2.5, 2.2) with 95% Confidence Interval
Mean = 0.80
95% CI Lower = 0.80
95% CI Upper = 0.80
Frequency
0
1
2
3
4
5
6
0.790
0.795
0.800
0.805
0.810
Utilization at Queue 0 for poisson-arrival(2.0), service_rate(2.5, 2.2)

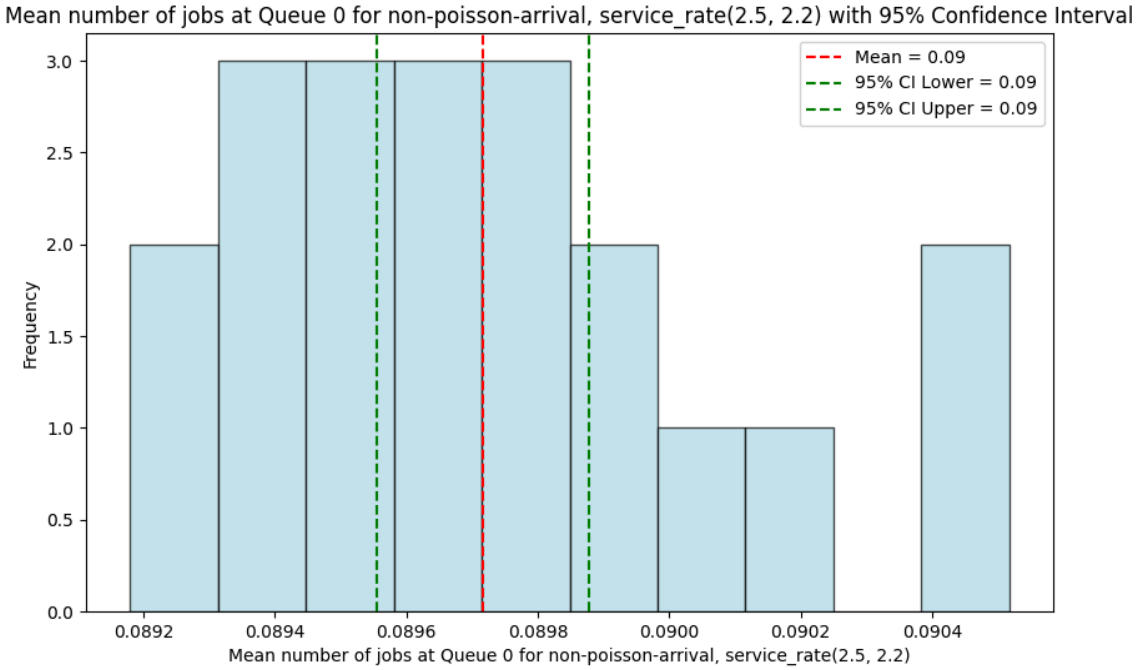

Utilization at Queue 0 for non-poisson-arrival, service_rate(2.5, 2.2) with 95% Confidence Interval
Mean = 0.09
95% CI Lower = 0.09
95% CI Upper = 0.09
Frequency
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.0892
0.0894
0.0896
0.0898
0.0900
0.0902
0.0904
Utilization at Queue 0 for non-poisson-arrival, service_rate(2.5, 2.2)

Metric: Mean number of Jobs in Queue0

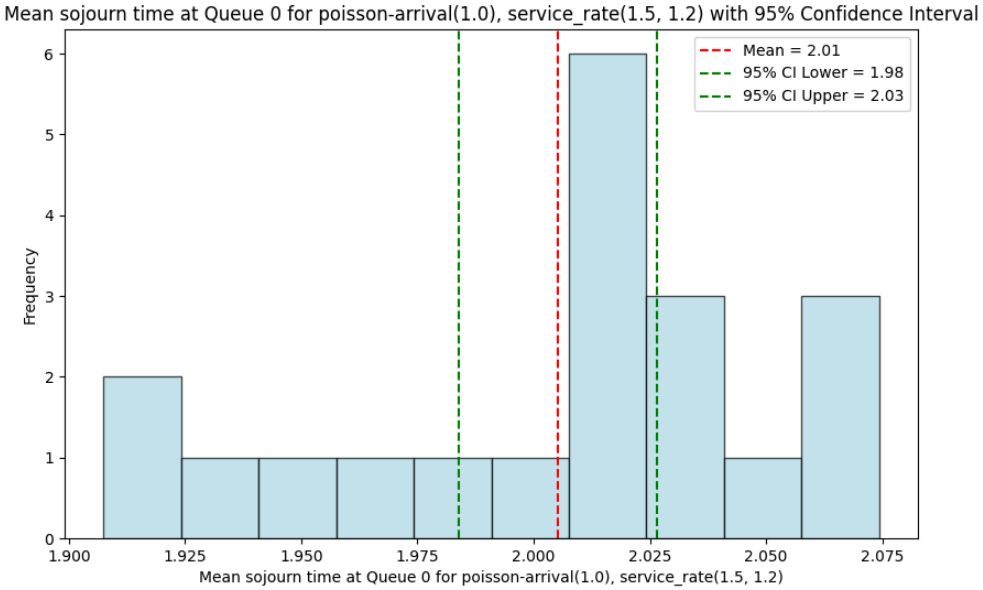


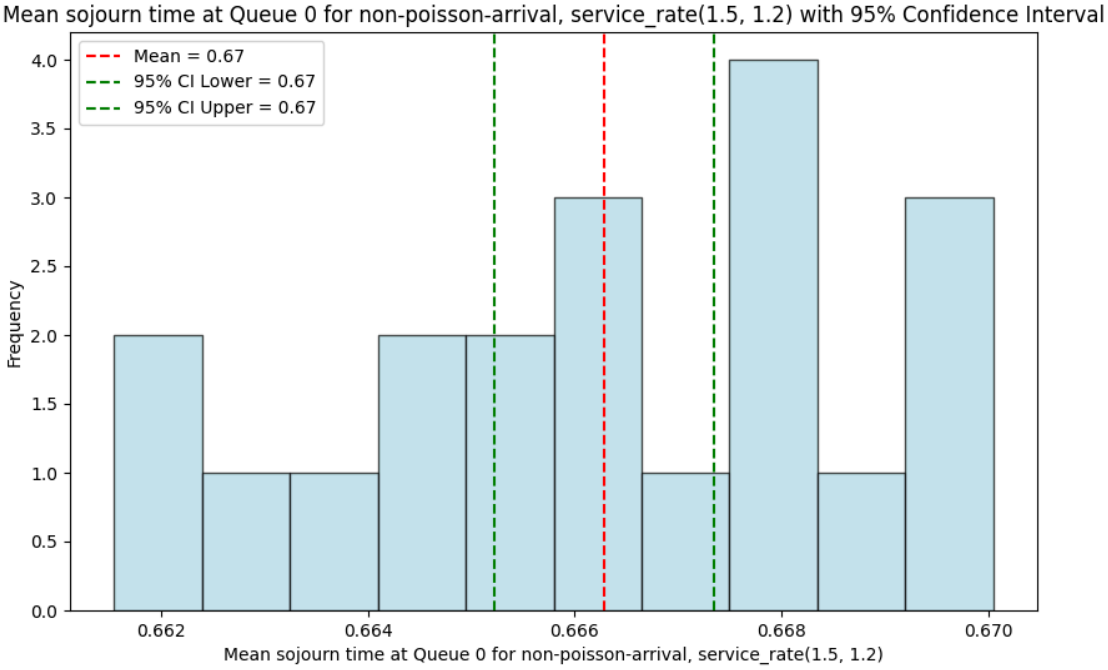


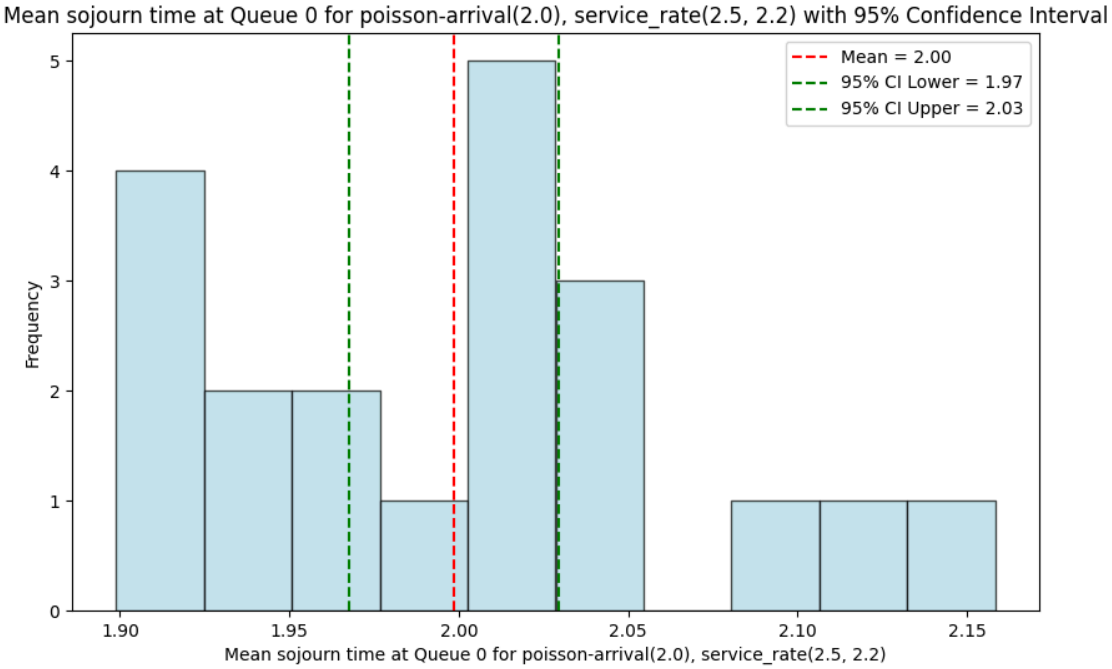

Mean number of jobs at Queue 0 for poisson-arrival(2.0), service_rate(2.5, 2.2) with 95% Confidence Interval
Mean = 3.99
95% CI Lower = 3.93
95% CI Upper = 4.06
Frequency
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
3.8
3.9
4.0
4.1
4.2
4.3
Mean number of jobs at Queue 0 for poisson-arrival(2.0), service_rate(2.5, 2.2)

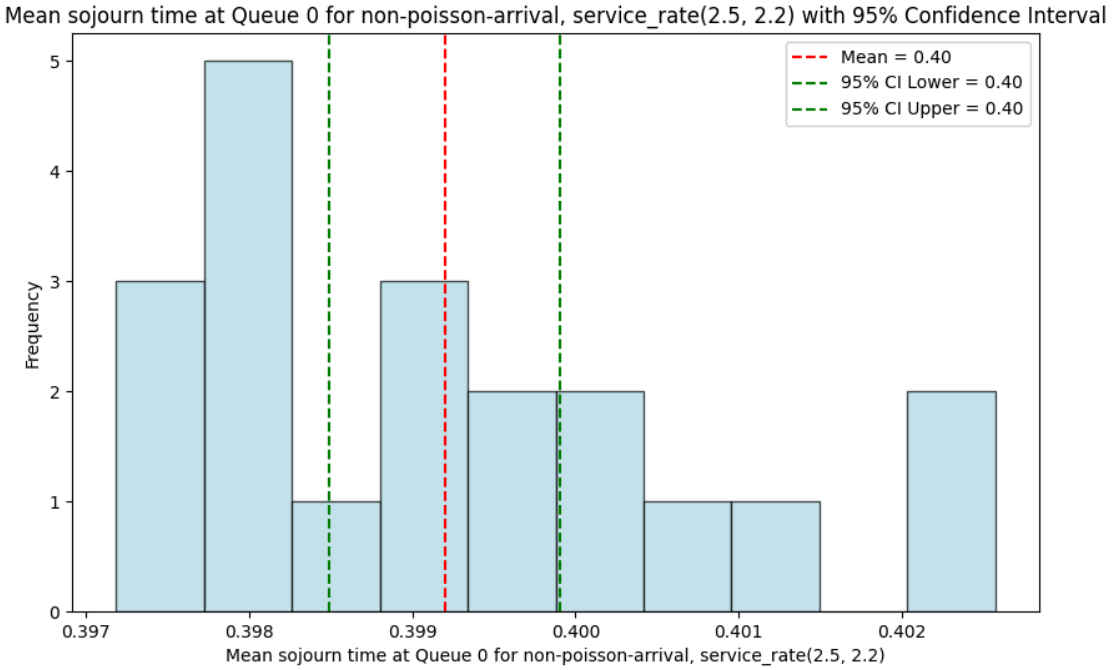

Mean number of jobs at Queue 0 for non-poisson-arrival, service_rate(2.5, 2.2) with 95% Confidence Interval
Mean = 0.09
95% CI Lower = 0.09
95% CI Upper = 0.09
Frequency
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0.0892
0.0894
0.0896
0.0898
0.0900
0.0902
0.0904
Mean number of jobs at Queue 0 for non-poisson-arrival, service_rate(2.5, 2.2)

Metric: Mean Soujourn time in Queue0






Mean sojourn time at Queue 0 for poisson-arrival(2.0), service_rate(2.5, 2.2) with 95% Confidence Interval
Mean = 2.00
95% CI Lower = 1.97
95% CI Upper = 2.03
Frequency
0
1
2
3
4
5
1.90
1.95
2.00
2.05
2.10
2.15
Mean sojourn time at Queue 0 for poisson-arrival(2.0), service_rate(2.5, 2.2)


Mean sojourn time at Queue 0 for non-poisson-arrival, service_rate(2.5, 2.2) with 95% Confidence Interval
Mean = 0.40
95% CI Lower = 0.40
95% CI Upper = 0.40
Frequency
0
1
2
3
4
5
0.397
0.398
0.399
0.400
0.401
0.402
Mean sojourn time at Queue 0 for non-poisson-arrival, service_rate(2.5, 2.2)