First Project - Airport Security Queues

Minerva University

CS166 - Modeling and Analysis of Complex Systems

Prof. Tambasco

February 2, 2025

1 Introduction (97)

Efficiently managing queues in airport security screening is a critical operational challenge. Airports must balance security manpower with incoming passengers while optimizing smooth screening to minimize waiting times. This project aims to investigate how different queue configurations impact waiting times, average queue lengths, and overall efficiency. Specifically, it will make a recommendation on the amount of airport security to ensure smooth operation. To find the best security station set-up, we will implement a simulation that models the airport security queueing system, analyze its performance under different conditions, and compare empirical results with theoretical predictions from queueing theory.

2 Theoretical Analysis

2.1 Model Description (183)

Models are simplified representations of real-world systems. They preserve key relationships and interactions between essential components while making assumptions and simplifications to facilitate analysis and manipulation. The analysis of airport security stations' configuration relies on the queueing theory, which provides a mathematical framework to analyze waiting lines with varied set-ups.

The security screening queue system involves travelers arriving at random intervals, joining the shortest available queue, and undergoing screening. Each security station serves one traveler at a time, and follows the first-come-first-out (FIFO) discipline. Additionally, some of the travelers require further screening. There is only one senior officer that can perform additional screening and everyone in the queue will wait until it finishes.

The parameters include arrival rate $\lambda = 10$ arrivals per minute, service rates $\mu = 30$ seconds per traveler, and $\mu_2 = 2$ minutes per traveler (for normal screening and additional screening). The airport opens all day everyday, thus there will not be breaks between services. Also, we assume the system has an infinite buffer, meaning the queues can grow infinitely long and the travelers will always wait until they are served.

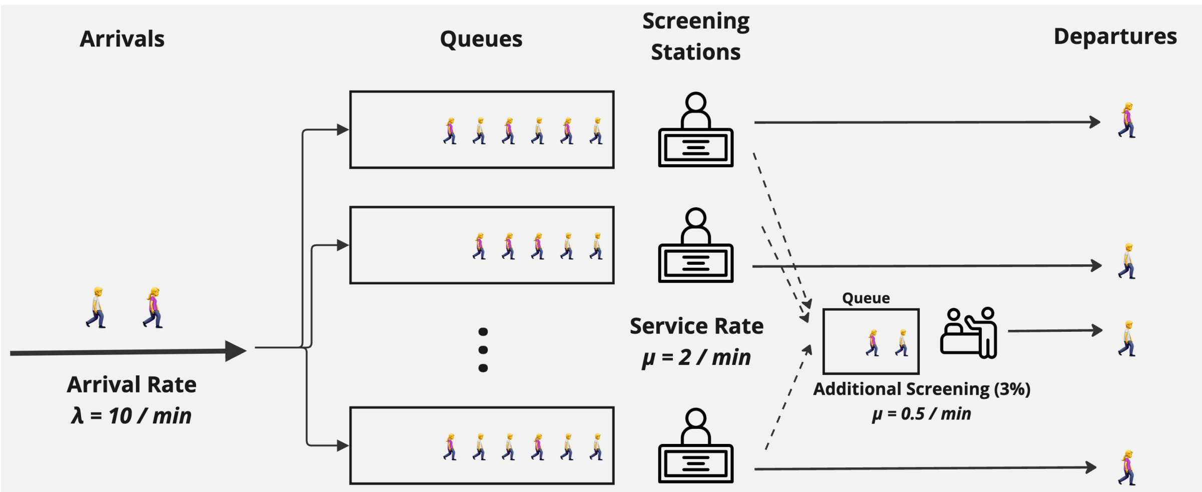

Figure 1: Conceptual diagram of the airport security screening queue system.(self-made)

2.2 Mathematical Framework

2.2.1 The Arrivals (156)

In the queueing theory, the travelers' arrival follows a Poisson process, meaning the inter-arrival times between travelers follow an exponential distribution. This brings about two important assumptions: each arrival is independent from another and the average rate λ does not change during the simulation. This makes the system memoryless, which means knowing when the last customer arrived provides no information about when the next customer will arrive.

The arrival rate (λ) is

$$\lambda = 10 \text{ travelers per minute,}$$

which corresponds to the scale parameter of the exponential distribution being $\beta = \frac{1}{\lambda} = \frac{1}{10}$. The probability density function for travelers arrival is:

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Some properties of the exponential distribution include the mean and the expected value of the inter-arrival time being $\frac{1}{\lambda} = \frac{1}{10}$ minutes, and the variance of $\frac{1}{\lambda^2}$ minutes.

2.2.2 The Queues (503)

After the travelers arrive at the airport, they join the shortest queue where they start the wait. There are multiple parallel security screening queues, each of which is an **M/G/1** queue. The **M** refers to *Markov Chain*, which is the assumption that arrivals are exponentially distributed and therefore memoryless. The **G** means *General*. In this case, we assume that the service time per traveler follows a Truncated Normal distribution. The final **1** means there is one service station per queue.

Suppose there are n security queues, the arrival rate for each queue is:

$$\lambda_1 = \frac{10}{n} \text{ travelers per minute.}$$

Each security screening station processes travelers one at a time, with a service time that follows a Truncated Normal distribution with a mean of 30 seconds ($\mu_1 = 30$) and a standard deviation of 10 seconds ($\sigma_1 = 10$). The truncated normal distribution in this case is cut off at 0 to avoid negative values. The probability density function is given by:

$$f(x;\mu_1,\sigma_1,a=0,b=\infty) = \begin{cases} \frac{\frac{1}{\sigma_1}\phi\left(\frac{x-\mu_1}{\sigma_1}\right)}{\Phi\left(\frac{b-\mu_1}{\sigma_1}\right)-\Phi\left(\frac{a-\mu_1}{\sigma_1}\right)}, & a \leq x \leq b, \\ 0, & \text{otherwise.} \end{cases}$$

where $\phi(\cdot)$ is the probability density function of the standard normal distribution and $\Phi(\cdot)$ is its cumulative distribution function.

Since the mean service time is 30 seconds (or 0.5 minutes) per traveler, the service rate (μ) is:

$$\mu_1 = \frac{1}{0.5} = 2 \text{ travelers per minute}$$

The utilization rate of each of the general queues (ρ_1) is given by:

$$\rho = \frac{\lambda_1}{\mu_1} = \frac{10}{n} \times \frac{1}{2} = \frac{5}{n}$$

This ratio helps determine whether queues are stable; a system is stable only if $\rho < 1$, meaning the number of service stations must satisfy $n > 5$. However, the stability of the system can be influenced by the additional screening process, too.

There is a 3% probability that a traveler will require additional screening. A senior officer will move around the general queues for those in need, and the process follows FIFO discipline. It can also be seen as an individual M/G/1 queue for computational clarity (and it does have properties of a queue). However, when the senior officer performs the additional screening, everyone in the queue from which the traveler originally comes will wait until they finish. Thus, this queue's behavior can influence the general queues' performance significantly.

The arrival rate to this queue is:

$$\lambda_2 = 0.03 \times 10 = 0.3 \text{ travelers per minute}$$

The service time follows a Truncated Normal distribution with a mean of 2 minutes ($\mu_2 = 2$) and a standard deviation of 2 minutes ($\sigma_2 = 2$). The corresponding service rate is:

$$\mu_2 = \frac{1}{2} = 0.5 \text{ travelers per minute}$$

Since there is only one queue (the senior security officer) handling additional screenings, the utilization rate (ρ_2) is:

$$\rho_2 = \frac{\lambda_2}{\mu_2} = \frac{0.3}{0.5} = 0.6$$

This utilization rate indicates that the additional screening queue is expected to be busy 60% of the time but should remain stable as long as $\rho_2 < 1$.

2.2.3 Measuring performance (41)

We will simulate the model in the next section. By comparing theoretical results with empirical simulations, we aim to validate the model's accuracy and determine the optimal number of security stations required for stable passenger flow, balancing efficiency with resource allocation.

3 Empirical Analysis

3.1 Simulation Design and Implementation (245)

3.1.1 Testing Code

The code for testing was first crafted, which includes several print*()* functions that show each step the system takes. This is to ensure that there is a working code available for simulation and that the code functions exactly how we want it to. The code is shown in Appendix A, following 4 test cases of scenarios in Appendix B.

3.1.2 Simulation Code

In Appendix C, all the *print()* functions are removed in the simulation code as the queueing system will be run many times, and we have already made sure that there is a functioning code capturing the dynamics of the system. Except for the *print()* functions, the testing code and simulation code are identical. The library *tqdm()* is implemented to track the progress of the simulation.

3.1.3 Simulation of A Day

The first simulation observes the dynamics of the airport security queueing system in a day. It aims to identify the optimal number of queues for reaching the equilibrium state by setting different numbers of queues in each simulation. It tracks the fluctuation of individual queue length and the waiting time of each traveler in a day. The code is in Appendix D.

3.1.4 Simulation of Key Performance Indicators

After identifying two possible optimal queue lengths, the second simulation measures the performance indicators: average queue length in a day, maximum queue length in a day, and average waiting time in queue. The code is in Appendix E.

3.2 Simulation Results and Discussion

3.2.1 Simulation of A Day (391 + 142)

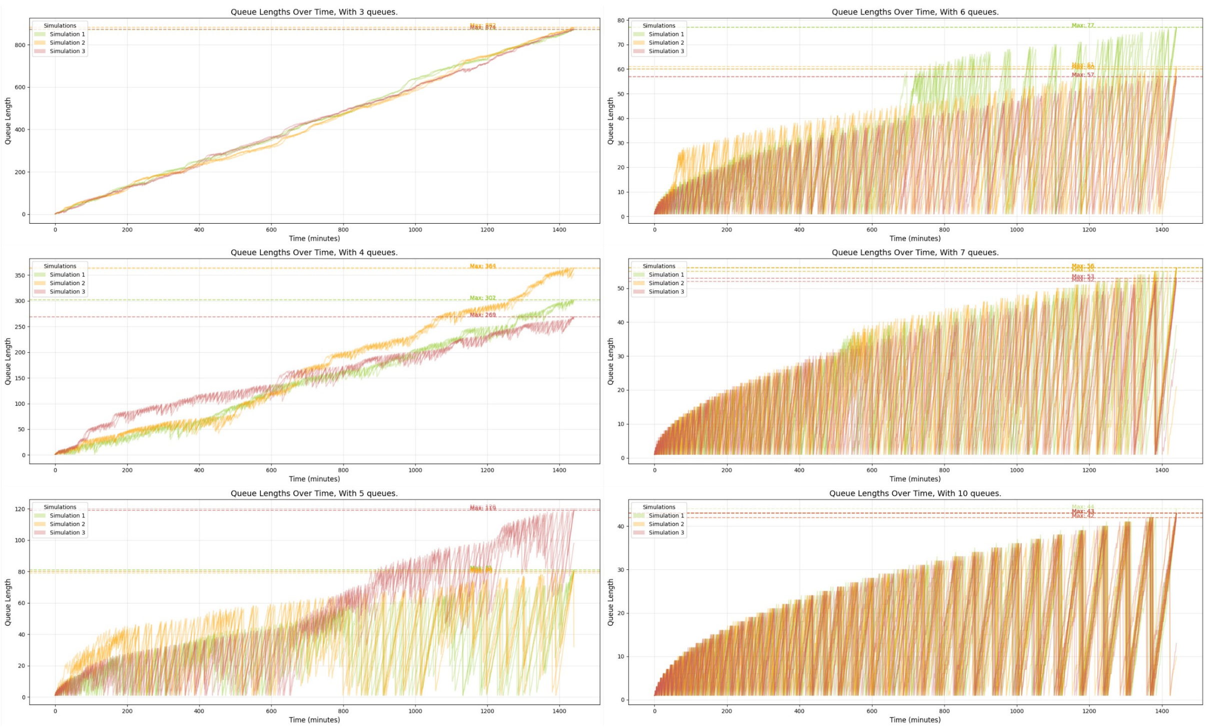

Figure 2: Queue length over time in a day. Please note that the y axis' higher limits are different for each graph, the y limits are adjusted this way to demonstrate the fluctuations within each system closely. The systems with fewer queues have higher maximum queue lengths in their oscillation.

As we calculated above, the utilization rate of the general screening queues is

$$\rho = \frac{\lambda_1}{\mu 1} = \frac{5}{n},$$

where n is the number of queues. A stable system requires $\rho < 1$, meaning that $n > 5$ is necessary. Figure 2 shows the queue length over time under three unstable settings (with 3, 4, and 5 queues) on the right and three stable settings (6, 7, and 10 queues) on the left. Figure 3 shows the wait time for each traveler with the same setting, the traveler number represents travelers' order to enter the airport system.

With 3 and 4 queues, we can see that the queue length grew linearly with the incoming travelers and the service was barely processing anyone out. This means the system was not able to reach an equilibrium and the queue length would grow indefinitely. The simulation with 4 queues had more oscillation because the queues were able to process some travelers but the arrivals were still overwhelming for the system.

With 5 queues, the oscillation increased largely in magnitude and the queue length went back to 0 from time to time. The 3 simulations (each with 5 queues) also has different traces. This shows that when utilization(ρ) is exactly 1, the system can sometimes be balanced (back to 0 queue length) but it is vulnerable to randomness given the arrival and service time are probabilistic and the 3% chance of additional screening.

With 6, 7, and 10 queues, the utilization rate is less than 1, meaning that the system is able to reach equilibrium. The systems' ability to balance arrival and service rates was reflected in their oscillation which always goes down to zero in every cycle. With 6 and 7 queues, some events significantly increased the queue length. This is due to the possibility of going into a long additional screening process. The system can function stably (without extremely long waiting times that significantly increase the queue length) and predictably when the number of queues reaches 10. When the possibility for additional screening was removed, the system was able to reach a stable state with 6 queues (see Figure 3).

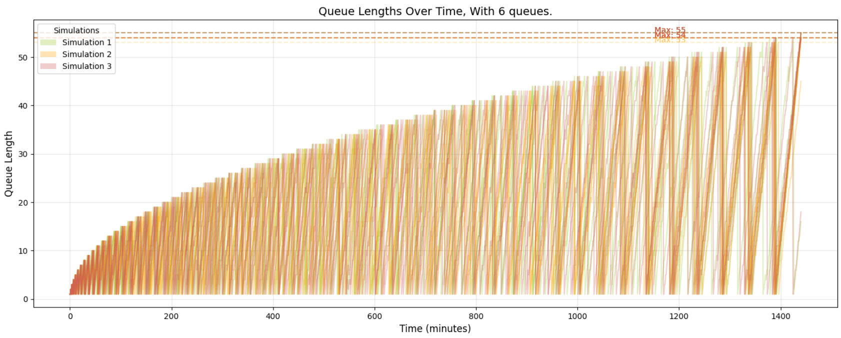

Figure 3: Wait time for each traveler in a day without additional screening.

The wait time plots (Figure 4) behave similarly as the queue length plots. With 3 to 5 queues, the queues were not balanced and the wait time eventually grew indefinitely. With 6 to 10 queues, the wait times have oscillations but always fall back to 0 in each cycle, showing stably functioning systems.

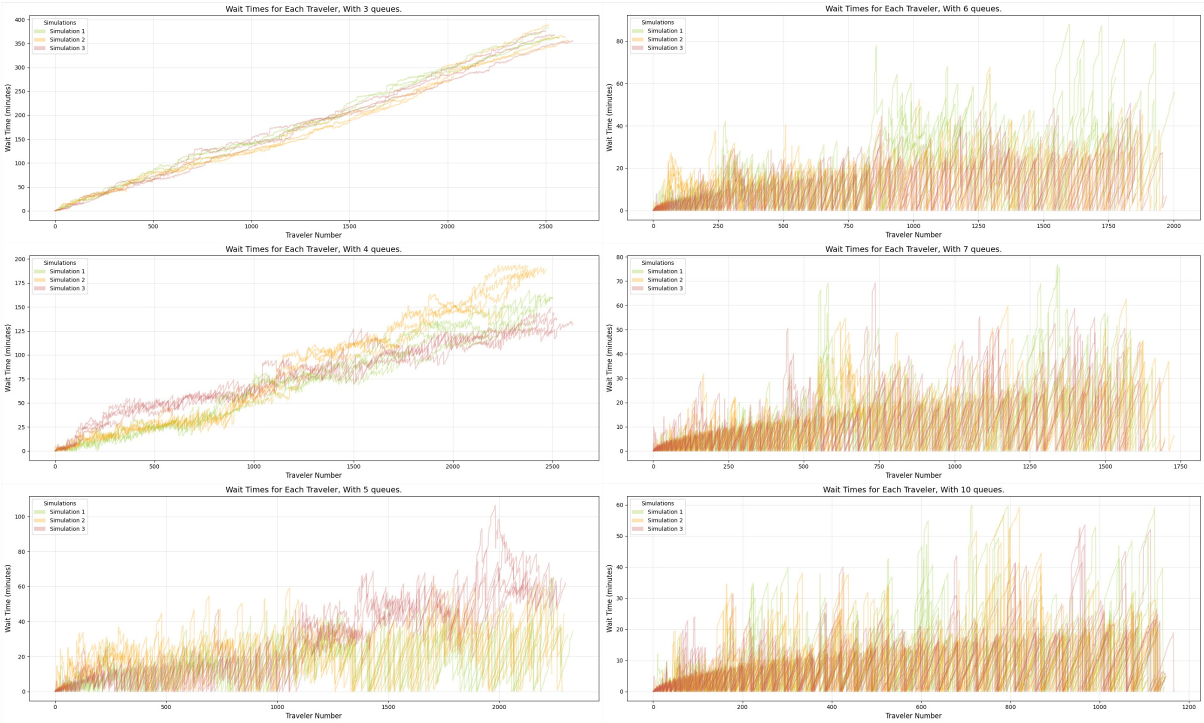

Figure 4: Wait time for each traveler in a day. Please note that the y axis' higher limits are different for each graph, y limits are adjusted this way to demonstrate the fluctuations within each system closely. The systems with fewer queues have higher maximum waiting times in their oscillation.

Based on the result of the simulation, I selected the queueing systems of 7 queues and 10 queues as possible choices for recommendation. I chose 7 queues because even though the system with 6 queues does reach equilibrium empirically and has $\rho < 1$ utilization rate theoretically, it behaves relatively unstably with some events significantly increasing the queue length. With 7 queues, the system was able to digest the extreme events better and faster. I chose 10 queues too because for 7 to 9 queues (for code to generate them, see Appendix D), there were still a few disturbances in the queueing system while for 10, the system digested the travelers stably without significant disturbance. This is shown in Figure 4, there are still very long individual wait times for the 10 queue systems but are not causing fluctuations of queue lengths in Figure 2.

3.2.2 Simulation of Key Performance Indicators (486)

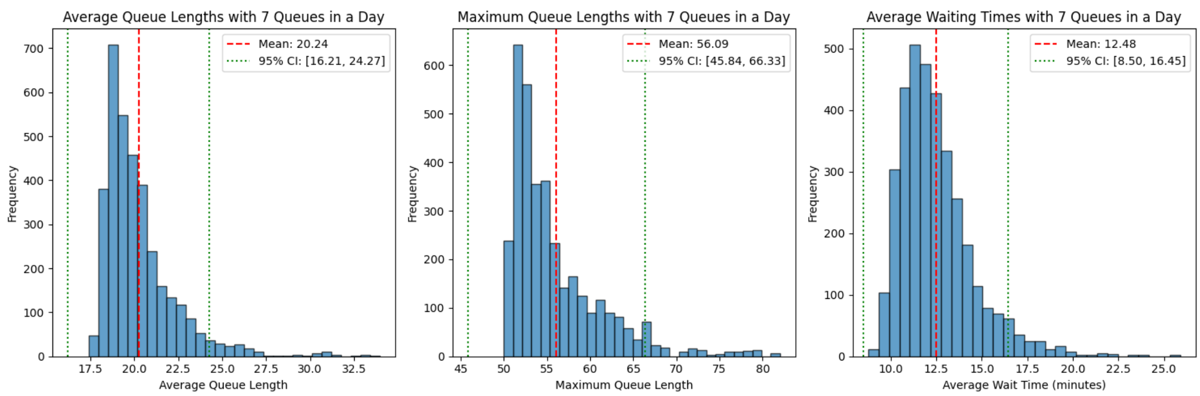

Figure 5: Performance Indicators for Queueing System with 7 Queues

For the queueing system with 7 queues, the histogram for the average queue length shows a right-skewed distribution, with the mode near 20 (people). The mean average queue length is 20.24, and the 95% confidence interval (CI) is [16.21, 24.27], indicating that the true average queue length is 95% likely to fall within this range. For the maximum queue length, the histogram similarly shows a right-skewed pattern, with an expected value (mean) of 56.09 and a 95% CI of [45.84, 66.33]. Lastly, the histogram for the average wait time reveals a mean of 12.48 minutes, and a 95% CI of [8.50, 16.45]. All three distributions are slightly skewed to the right, which aligns with the probabilistic nature of the arrivals and the memoryless property of the Poisson process influencing the queue system.

Little's Law is a theorem in queueing theory that states the average number of customers in a system (L) is equal to the product of the average arrival rate (λ) and the average time a customer spends in the system (W), expressed as

$$L = \lambda W,$$

disregarding the arrival and service distribution. For this system, where the arrival rate λ is $10/7 \approx 1.43$ travelers per minute and the average waiting time W is 12.48 minutes, the theoretical average queue length L is $1.43 \times 12.48 \approx 17.83$. This value is lower than the empirical mean of $L = 20.24$. The differences can be caused by random delays caused by variability in arrival and service times, and the impact of the senior officer handling additional screenings that require longer service time.

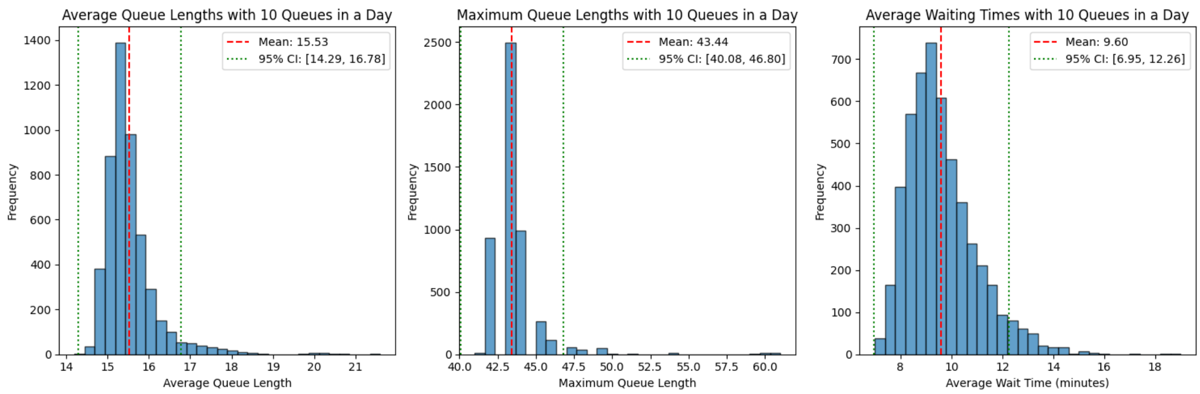

Figure 6: Performance Indicators for Queueing System with 10 Queues

The performance of the queueing system with 10 queues exhibits better efficiency. The histogram for the average queue length shows a tighter concentrated distribution around a mean of 15.53. The 95% CI of [14.29, 16.78] indicates that we have narrowed down the range for the true population mean and increased the system's predictability, which aligns with the observation that 10 queue systems are more stable. The maximum queue length has a mean of 43.44, a 95% CI of [40.08, 46.80]. The average waiting time is also reduced, with a mean of 9.6 minutes and a 95% CI of [6.95, 12.26]. These metrics indicate that the system with 10 queues is better balanced and operates with minimal fluctuations.

Applying Little's Law to this system, which has an arrival rate λ of $10/10 = 1$ traveler per minute and an average waiting time W of 9.60 minutes, the theoretical queue length L is $1.0 \times 9.60 = 9.60$. This theoretical value is also lower than the empirical mean of $L = 15.53$ due to the same factors affecting the 7 queue system: random variation in arrivals and service times, and the delays caused by the additional screening process.

4 Recommendations

With theoretical and empirical analyses, this report has explored how queue configurations affect the efficiency and stability of an airport security screening system. Theoretical calculations found out that at least 5 queues are required to maintain equilibrium, and empirical simulations confirmed the system's ability to reach a relatively stable equilibrium state under varying queue setups. However, while increasing the number of queues enhances efficiency by reducing wait times and queue lengths, resource allocation should also be considered.

Based on the findings, the recommendation is to implement 7 queues, which provide a balanced trade-off between efficiency and resource constraints. The empirical analysis showed that this configuration maintains a fairly stable system, with an average waiting time of 12.48 minutes and a maximum queue length of about 56—both of which are reasonable given that most travelers have buffer time for check-in and security screening. Although the 10-queue system showcases slightly better performance, the additional costs may not justify the marginal gains.

Word Count: 2404

AI Statement

I used Grammarly to help correct my grammar mistakes. I used Claude to help debug and correct my code in Python. No other AI tools were used.

Collaboration Statement

I did not discuss, cross-check, or collaborate with anyone on this assignment.

5 Reference

- CS166. Lesson 3 Pre-Class resources -for code implements an M/M/1 queue
- CS166. Lesson 2. Pre-Class Work - for theoretical analysis of the queueing theory
- Wikipedia - Little's law -for explanations on Little's law
- Wikipedia - Exponential distribution -for explanations on parameters and PDF of Exponential distribution
- Wikipedia - Exponential distribution -for explanations on parameters and PDF of Exponential distribution
- Investopedia - Queuing Theory Definition, Elements, and Example -for detailed explanations on the Queueing theory
- scipy documentation - scipy.stats.truncnorm -for the syntax and parameters of creating a truncated normal distribution in Python
- Overleaf guides - Mathematical expressions

6 Appendix

6.1 Appendix A - Testing Code

```python
import heapq
import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt

class AdditionalScreening():
    '''
    Represents additional screening handled by a single senior officer (shared amoung queues)
    The officer inspects travelers who trigger security alerts (3% chance).
    '''

    def __init__(self, screening_distribution):
        self.screening_queue = []  # Tracks travelers who need additional screening
        self.screening_time = [] # Record how long each passenger will be screened
        self.screening_distribution = screening_distribution #The screening time distribution
        self.next_departure_time = np.inf  # When the officer is done screening

    def add_traveler(self, arrival_time):
        '''
        Adds a traveler to the queue and record the screening time they need.

        We generate and record screening time for each new traveler so we can
        calculate the expected time they will depart (their arrival time + all
        the screening time travelers in front of them need) and report to their queue.
        '''
        # Push the item arrival time to the priority queue
        heapq.heappush(self.screening_queue, arrival_time)
        # Push the needed service time for this traveler to the priority queue
        heapq.heappush(self.screening_time, self.screening_distribution.rvs())

        # Checkpoint
        print(f"Traveler at {arrival_time} requires additional screening! There are {len(self.screening_queue)-1} people in the line and this will take {sum(self.screening_time)} minutes!")

        # If we went from an empty queue to 1 person
        if len(self.screening_queue) == 1:
            # Sanity check
            assert self.next_departure_time == np.inf
            # Generate the next departure time (because it is currently infinity).
            self.next_departure_time = arrival_time + self.screening_time[0]
```

```python

    def serve_traveler(self):
        '''
        Pops out a traveler from the front of the queue and remove their recorded screening time.
        '''

        # Remove the traveler that is screened
        heapq.heappop(self.screening_queue)
        # Remove their correspondent screening time
        heapq.heappop(self.screening_time)

        if len(self.screening_queue) == 0:
            # The queue is empty so we should not generate a new departure time.
            self.next_departure_time = np.inf
        else:
            # Generate the next departure time
            self.next_departure_time += self.screening_time[0]

    def request_screening(self, arrival_time):
        '''
        Adds the new traveler to the queue and check if there is any departures.
        '''
        if arrival_time < self.next_departure_time:
            # Handle arrivals
            self.add_traveler(arrival_time)
        else:
            # Handle departures
            while arrival_time > self.next_departure_time:
                self.serve_traveler()

        # Sanity check
        assert len(self.screening_queue) >= 0

        # Return the time the traveler needs to wait for them and every one in front
        # of them to finish the additional screening to their queue
        return sum(self.screening_time)

class Queue:
    def __init__(self, service_distribution, senior_officer):
        self.priority_queue = []  # Tracks people in the queue
        self.service_distribution = service_distribution # The service time distribution
        self.senior_officer = senior_officer # Pass in access to AdditionalScreening()
        self.next_service_time = 0 # The time when the previous person departs and the next is being served
        self.next_departure_time = np.inf  # Tracks when travelers depart
```

```python

        # For simuation purpose
        self.queue_length_history = [] # Tracks the queue length at every arrival
        self.wait_time_history = [] # Tracks the waiting time for each traveler
        self.run_until = np.inf # Update when we receive the length for simulation (when to stop)

    def add_traveler(self, arrival_time):
        '''
        Adds a traveler to the queue and starts service if idle.
        '''

        # Checkpoint
        print(f"New arrival at {arrival_time}!")

        # Push the traveler (by their arrival time) to the priority queue
        heapq.heappush(self.priority_queue, arrival_time)

        # Record the current queue length
        self.queue_length_history.append(len(self.priority_queue))

        if len(self.priority_queue) == 1:
            # If we went from an empty queue to 1 person and immidiately start serving
            # Generate the next departure time (because it is currently infinity).

            assert self.next_departure_time == np.inf  # Sanity check
            self.next_service_time = arrival_time # Immidiately start serving
            self.next_departure_time = arrival_time + self.service_distribution.rvs()

    def serve_traveler(self):
        '''
        Pops out a traveler from the front of the queue and record their waiting time.
        '''

        # Remove the traveler from the front of the queue
        arrival_time = heapq.heappop(self.priority_queue)
        # Record the time they have waited
        self.wait_time_history.append(self.next_service_time - arrival_time)

        # Checkpoint
        print(f"Start serving at {self.next_service_time}, this traveler waited {self.next_service_time - arrival_time} minutes!")

        # 3% chance of needing additional screening
        if np.random.rand() < 0.03:
            # Pass the traveler and their original departure time (= their arrival time in the
```

```
            # additional screening queue) to request_screening() function. And update the departure
            # time of the traveler after their inspection
            self.next_departure_time += self.senior_officer.request_screening(self.next_departure_time)

        # Stop serving if the next departure time will exceed time of simulation
        if self.next_departure_time > self.run_until:
            return

        # Checkpoint
        print(f"New departure at {self.next_departure_time}!")

        if len(self.priority_queue) == 0:
            # The queue is empty so we should not generate a new departure time.
            self.next_departure_time = np.inf
            # Checkpoint
            print(f"Queue empty!")

        else:
            # One travler departs and immidiately start serving the next
            self.next_service_time = self.next_departure_time
            # Generate the next departure time
            self.next_departure_time += self.service_distribution.rvs()

    def join_queue(self, arrival_time):
        '''
        Adds the new traveler to the queue and check if there is any departures.
        '''
        if arrival_time < self.next_departure_time:
            # Handle arrivals
            self.add_traveler(arrival_time)
        else:
            # Handle departures
            while arrival_time > self.next_departure_time:
                if self.next_departure_time > self.run_until:
                    print(f"This queue stopped because the next departure time is {self.next_departure_time}.")
                    return
                self.serve_traveler()

        assert len(self.priority_queue) >= 0  # Sanity check

class Airport:
    def __init__(self, arrival_distribution, service_distribution, additional_screening_distribution, num_of_queues):
```

```python
        # Pass in distributions of arrival, service, and additional screening time
        self.arrival_distribution = arrival_distribution
        self.service_distribution = service_distribution
        self.additional_screening_distribution = additional_screening_distribution

        # Initiate 1 additional screening queue
        self.senior_officer = AdditionalScreening(additional_screening_distribution)
        # Initiate n Queue()
        self.queues = [Queue(service_distribution, self.senior_officer) for _ in range(num_of_queues)]

    def get_least_busy_queue_index(self):
        '''
        Returns the index of the queue with the least elements in priority_queue.
        '''
        # Generate a list contains the length of the priority_queue in each Queue()
        queue_lengths = [len(queue.priority_queue) for queue in self.queues]

        # Return the index of the minimun
        return np.argmin(queue_lengths)

    def handle_arrival(self, arrival_time):
        '''
        Handles the arrival of a traveler by placing them in the least busy queue.
        '''
        # Get the index of the least busy queue
        least_busy_index = self.get_least_busy_queue_index()
        # Add the traveler into the queue
        self.queues[least_busy_index].join_queue(arrival_time)

        # Checkpoint
        print(f"Traveler joined queue {least_busy_index}.")

    def get_queue_histories(self):
        '''
        Returns two lists containing queue_length_history and wait_time_history for each queue.
        '''
        # Generate a list contains the history of queue length in each Queue()
        queue_lengths = [queue.queue_length_history for queue in self.queues]
        # Generate a list contains the record of waiting time in each Queue()
        wait_times = [queue.wait_time_history for queue in self.queues]
        return queue_lengths, wait_times

    def run(self, run_until):
        '''
```

```
        Runs the simulation until the specified time.
        Returns two lists containing queue_length_history and wait_time_history for each queue.
        '''
        # Update when to stop the simulation for each queue
        for queue in self.queues:
            queue.run_until = run_until

        # Generate the first arrival
        new_arrival_time = self.arrival_distribution.rvs()

        # Check if we ran out of time
        while new_arrival_time < run_until:
            self.handle_arrival(new_arrival_time) # Accept the new arrival
            new_arrival_time += self.arrival_distribution.rvs() # Generate a new arrival time

        # Checkpoint
        print(f'Stopped because next arrival time will be {new_arrival_time}.')

        # Return two lists (queue length and waiting time) for further analysis
        return self.get_queue_histories()
```

6.2 Appendix B - Testing Scenarios

```
## Test case 1. -- with only 1 queue (so the timeline looks logical)

# Arrival rate and Service rate
arrival_rate = 10 #10 travelers per minute
service_rate = 2 #2 travelers per minute

## For Queue()
# Parameters for the service time distribution (truncated normal)
mean_service = 1/service_rate
sigma_service = 1/6
a_service = (0 - mean_service) / sigma_service  # Standardized lower bound (0)
b_service = np.inf  # No upper bound (infinity)

## For AdditionalScreening()
# Parameters for the additional screening time distribution (truncated normal)
mean_as = 2
sigma_as = 2
a_as = (0 - mean_as) / sigma_as  # Standardized lower bound (0)
b_as = np.inf  # No upper bound (infinity)

# Create distributions
```

```python
# Exponential inter-arrival time
arrival_distribution = sts.expon(scale=1/arrival_rate)
# Truncated normal service time
service_distribution = sts.truncnorm(a_service, b_service, loc=mean_service,
scale=sigma_service)
# Truncated normal additional screening time
additional_screening_distribution = sts.truncnorm(a_as, b_as, loc=mean_as, scale=2)
num_of_queues = 1

# Create Airport() object
airport = Airport(arrival_distribution, service_distribution,
additional_screening_distribution, num_of_queues)

# Run simulation
print(airport.run(10))
```

```python
## Test case 2. -- very short time period (see if the system will stop itself when time runs
out)

# Arrival rate and Service rate
arrival_rate = 10 #10 travelers per minute
service_rate = 4 #4 travelers per minute

## For Queue()
# Parameters for the service time distribution (truncated normal)
mean_service = 1/service_rate
sigma_service = 1/6
a_service = (0 - mean_service) / sigma_service  # Standardized lower bound
b_service = np.inf  # No upper bound (infinity)

## For AdditionalScreening()
# Parameters for the additional screening time distribution (truncated normal)
mean_as = 2
sigma_as = 2
a_as = (0 - mean_as) / sigma_as  # Standardized lower bound
b_as = np.inf  # No upper bound (infinity)

# Create distributions
# Exponential inter-arrival time
arrival_distribution = sts.expon(scale=1/arrival_rate)
# Truncated normal service time
service_distribution = sts.truncnorm(a_service, b_service, loc=mean_service,
scale=sigma_service)
# Truncated normal additional screening time
additional_screening_distribution = sts.truncnorm(a_as, b_as, loc=mean_as, scale=2)
num_of_queues = 3
```

```
# Create Airport() object
airport = Airport(arrival_distribution, service_distribution,
additional_screening_distribution, num_of_queues)

# Run simulation
print(airport.run(.5))
```

```
## Test case 3. -- When arrival_rate is a lot smaller than service_rate --> Little
travelers and fast service

# Arrival rate and Service rate
arrival_rate = 3 #3 travelers per minute
service_rate = 6 #6 travelers per minute

## For Queue()
# Parameters for the service time distribution (truncated normal)
mean_service = 1/service_rate #1/6 minute per traveler
sigma_service = 1/6
a_service = (0 - mean_service) / sigma_service  # Standardized lower bound
b_service = np.inf  # No upper bound (infinity)

## For AdditionalScreening()
# Parameters for the additional screening time distribution (truncated normal)
mean_as = 2
sigma_as = 2
a_as = (0 - mean_as) / sigma_as  # Standardized lower bound
b_as = np.inf  # No upper bound (infinity)

# Create distributions
# Exponential inter-arrival time
arrival_distribution = sts.expon(scale=1/arrival_rate)
# Truncated normal service time
service_distribution = sts.truncnorm(a_service, b_service, loc=mean_service,
scale=sigma_service)
# Truncated normal additional screening time
additional_screening_distribution = sts.truncnorm(a_as, b_as, loc=mean_as, scale=2)
num_of_queues = 3

# Create Airport() object
airport = Airport(arrival_distribution, service_distribution,
additional_screening_distribution, num_of_queues)

# Run simulation
print(airport.run(10))
```

```python
# Test case 4. -- When arrival_rate is a lot larger than service_rate --> Lots of travelers and slow service

# Arrival rate and Service rate
arrival_rate = 12 #12 travelers per minute
service_rate = .5 #.5 travelers per minute

## For Queue()
# Parameters for the service time distribution (truncated normal)
mean_service = 1/service_rate #2 mins per traveler
sigma_service = 1/6
a_service = (0 - mean_service) / sigma_service  # Standardized lower bound
b_service = np.inf  # No upper bound (infinity)

## For AdditionalScreening()
# Parameters for the additional screening time distribution (truncated normal)
mean_as = 2
sigma_as = 2
a_as = (0 - mean_as) / sigma_as  # Standardized lower bound
b_as = np.inf  # No upper bound (infinity)

# Create distributions
# Exponential inter-arrival time
arrival_distribution = sts.expon(scale=1/arrival_rate)
# Truncated normal service time
service_distribution = sts.truncnorm(a_service, b_service, loc=mean_service, scale=sigma_service)
# Truncated normal additional screening time
additional_screening_distribution = sts.truncnorm(a_as, b_as, loc=mean_as, scale=2)
num_of_queues = 3

# Create Airport() object
airport = Airport(arrival_distribution, service_distribution, additional_screening_distribution, num_of_queues)

# Run simulation
print(airport.run(10))
```

6.3 Appendix C - Simulation Code

```python
import heapq
import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt

class AdditionalScreening():
```

```
    '''
    Represents additional screening handled by a single senior officer (shared amoung queues)
    The officer inspects travelers who trigger security alerts (3% chance).
    '''

    def __init__(self, screening_distribution):
      self.screening_queue = []  # Tracks travelers who need additional screening
      self.screening_time = [] # Record how long each passenger will be screened
      self.screening_distribution = screening_distribution #The screening time distribution
      self.next_departure_time = np.inf  # When the officer is done screening

    def add_traveler(self, arrival_time):
      '''
      Adds a traveler to the queue and record the screening time they need.

      We generate and record screening time for each new traveler so we can
      calculate the expected time they will depart (their arrival time + all
      the screening time travelers in front of them need) and report to their queue.
      '''
      # Push the item arrival time to the priority queue
      heapq.heappush(self.screening_queue, arrival_time)
      # Push the needed service time for this traveler to the priority queue
      heapq.heappush(self.screening_time, self.screening_distribution.rvs())

      # If we went from an empty queue to 1 person
      if len(self.screening_queue) == 1:
        # Sanity check
        assert self.next_departure_time == np.inf
        # Generate the next departure time (because it is currently infinity).
        self.next_departure_time = arrival_time + self.screening_time[0]

    def serve_traveler(self):
      '''
      Pops out a traveler from the front of the queue and remove their recorded screening time.
      '''

      # Remove the traveler that is screened
      heapq.heappop(self.screening_queue)
      # Remove their correspondent screening time
      heapq.heappop(self.screening_time)

      if len(self.screening_queue) == 0:
        # The queue is empty so we should not generate a new departure time.
```

```python
            self.next_departure_time = np.inf
        else:
            # Generate the next departure time
            self.next_departure_time += self.screening_time[0]

    def request_screening(self, arrival_time):
        '''
        Adds the new traveler to the queue and check if there is any departures.
        '''
        if arrival_time < self.next_departure_time:
            # Handle arrivals
            self.add_traveler(arrival_time)
        else:
            # Handle departures
            while arrival_time > self.next_departure_time:
                self.serve_traveler()

        # Sanity check
        assert len(self.screening_queue) >= 0

        # Return the time the traveler needs to wait for them and every one in front
        # of them to finish the additional screening to their queue
        return sum(self.screening_time)

class Queue:
    def __init__(self, service_distribution, senior_officer):
        self.priority_queue = []  # Tracks people in the queue
        self.service_distribution = service_distribution # The service time distribution
        self.senior_officer = senior_officer # Pass in access to AdditionalScreening()
        self.next_service_time = 0 # The time when the previous person departs and the next is being served
        self.next_departure_time = np.inf  # Tracks when travelers depart

        # For simuation purpose
        self.queue_length_history = [] # Tracks the queue length at every arrival
        self.wait_time_history = [] # Tracks the waiting time for each traveler
        self.run_until = np.inf # Update when we receive the length for simulation (when to stop)

    def add_traveler(self, arrival_time):
        '''
        Adds a traveler to the queue and starts service if idle.
        '''

        # Push the traveler (by their arrival time) to the priority queue
        heapq.heappush(self.priority_queue, arrival_time)
```

```python

        # Record the current queue length
        self.queue_length_history.append(len(self.priority_queue))

        if len(self.priority_queue) == 1:
            # If we went from an empty queue to 1 person and immidiately start serving
            # Generate the next departure time (because it is currently infinity).

            self.next_service_time = arrival_time # Immidiately start serving
            self.next_departure_time = arrival_time + self.service_distribution.rvs()

    def serve_traveler(self):
        '''
        Pops out a traveler from the front of the queue and record their waiting time.
        '''

        # Remove the traveler from the front of the queue
        arrival_time = heapq.heappop(self.priority_queue)
        # Record the time they have waited
        self.wait_time_history.append(self.next_service_time - arrival_time)

        # 3% chance of needing additional screening
        if np.random.rand() < 0.03:
            # Pass the traveler and their original departure time (= their arrival time in the
            # additional screening queue) to request_screening() function. And update the departure
            # time of the traveler after their inspection
            self.next_departure_time += self.senior_officer.request_screening(self.next_departure_time)

        # Stop serving if the next departure time will exceed time of simulation
        if self.next_departure_time > self.run_until:
            return

        if len(self.priority_queue) == 0:
            # The queue is empty so we should not generate a new departure time.
            self.next_departure_time = np.inf

        else:
            # One travler departs and immidiately start serving the next
            self.next_service_time = self.next_departure_time
            # Generate the next departure time
            self.next_departure_time += self.service_distribution.rvs()

    def join_queue(self, arrival_time):
        '''
```

```python
        Adds the new traveler to the queue and check if there is any departures.
        '''
        if arrival_time < self.next_departure_time:
            # Handle arrivals
            self.add_traveler(arrival_time)
        else:
            # Handle departures
            while arrival_time > self.next_departure_time:
                if self.next_departure_time > self.run_until:
                    return
                self.serve_traveler()

        assert len(self.priority_queue) >= 0  # Sanity check

class Airport:
    def __init__(self, arrival_distribution, service_distribution,
    additional_screening_distribution, num_of_queues):
        # Pass in distributions of arrival, service, and additional screening time
        self.arrival_distribution = arrival_distribution
        self.service_distribution = service_distribution
        self.additional_screening_distribution = additional_screening_distribution

        # Initiate 1 additional screening queue
        self.senior_officer = AdditionalScreening(additional_screening_distribution)
        # Initiate n Queue()
        self.queues = [Queue(service_distribution, self.senior_officer) for _ in
        range(num_of_queues)]

    def get_least_busy_queue_index(self):
        '''
        Returns the index of the queue with the least elements in priority_queue.
        '''
        # Generate a list contains the length of the priority_queue in each Queue()
        queue_lengths = [len(queue.priority_queue) for queue in self.queues]

        # Return the index of the minimun
        return np.argmin(queue_lengths)

    def handle_arrival(self, arrival_time):
        '''
        Handles the arrival of a traveler by placing them in the least busy queue.
        '''
        # Get the index of the least busy queue
        least_busy_index = self.get_least_busy_queue_index()
        # Add the traveler into the queue
        self.queues[least_busy_index].join_queue(arrival_time)
```

```
    def get_queue_histories(self):
        '''
        Returns two lists containing queue_length_history and wait_time_history for each queue.
        '''
        # Generate a list contains the history of queue length in each Queue()
        queue_lengths = [queue.queue_length_history for queue in self.queues]
        # Generate a list contains the record of waiting time in each Queue()
        wait_times = [queue.wait_time_history for queue in self.queues]
        return queue_lengths, wait_times

    def run(self, run_until):
        '''
        Runs the simulation until the specified time.
        Returns two lists containing queue_length_history and wait_time_history for each queue.
        '''
        # Update when to stop the simulation for each queue
        for queue in self.queues:
            queue.run_until = run_until

        # Generate the first arrival
        new_arrival_time = self.arrival_distribution.rvs()

        # Check if we ran out of time
        while new_arrival_time < run_until:
            self.handle_arrival(new_arrival_time) # Accept the new arrival
            new_arrival_time += self.arrival_distribution.rvs() # Generate a new arrival time

        # Return two lists (queue length and waiting time) for further analysis
        return self.get_queue_histories()
```

6.4 Appendix D - Simulation of A Day

```
# Plotting the fluctuation of queue length and waiting time in each simulation

import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt
from matplotlib.patches import Patch

def run_and_plot_simulations(num_simulations, run_until, num_queues):
    # Parameters
    arrival_rate = 10
```

```
service_rate = 2

# Service time distribution parameters
mean_service = 1/service_rate
sigma_service = 1/6
a_service = (0 - mean_service) / sigma_service
b_service = np.inf

# Additional screening parameters
mean_as = 2
sigma_as = 2
a_as = (0 - mean_as) / sigma_as
b_as = np.inf

# Create distributions
arrival_distribution = sts.expon(scale=1/arrival_rate)
service_distribution = sts.truncnorm(a_service, b_service, loc=mean_service,
scale=sigma_service)
additional_screening_distribution = sts.truncnorm(a_as, b_as, loc=mean_as,
scale=sigma_as)

# Create figure with subplots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(15, 12))

# Colors for different simulations
colors = ['yellowgreen', 'orange', 'indianred']

# Store maximum queue lengths
max_lengths = []

# Create legend elements for simulations
legend_elements = [Patch(facecolor=color, alpha=0.3, label=f'Simulation {i+1}')
          for i, color in enumerate(colors)]

for sim in range(num_simulations):
  print(f"Running simulation {sim + 1}...")

  # Create new airport instance
  airport = Airport(arrival_distribution, service_distribution,
          additional_screening_distribution, num_queues)

  # Run simulation
  queue_lengths, wait_times = airport.run(run_until)

  # Plot queue lengths for each queue
  for q in range(num_queues):
```

```
        # Create time points (one for each length measurement)
        time_points = np.linspace(0, run_until, len(queue_lengths[q]))

        # Plot queue lengths
        line = ax1.plot(time_points, queue_lengths[q],
                       alpha=0.3,
                       color=colors[sim])

        # Store and plot maximum
        max_length = np.max(queue_lengths[q])
        max_lengths.append(max_length)
        ax1.axhline(y=max_length,
                    color=colors[sim],
                    linestyle='--',
                    alpha=0.2)

        # Plot wait times
        ax2.plot(range(len(wait_times[q])),
                 wait_times[q],
                 alpha=0.3,
                 color=colors[sim])

# Add simulation color legend to both plots
ax1.legend(handles=legend_elements, loc='upper left', title='Simulations')
ax2.legend(handles=legend_elements, loc='upper left', title='Simulations')

# Customize queue length plot
ax1.set_title(f'Queue Lengths Over Time, With {num_queues} queues.', fontsize=14)
ax1.set_xlabel('Time (minutes)', fontsize=12)
ax1.set_ylabel('Queue Length', fontsize=12)
ax1.set_ylim(0,400)

ax1.grid(True, alpha=0.3)

# Add text annotations for maximum queue lengths
for i, max_len in enumerate(max_lengths):
    sim_num = i // num_queues + 1
    queue_num = i % num_queues + 1
    ax1.text(run_until * 0.8, max_len,
            f'Max: {max_len:.0f}',
            color=colors[(i // num_queues)],
            alpha=0.7)

# Customize wait times plot
ax2.set_title(f'Wait Times for Each Traveler, With {num_queues} queues.',
fontsize=14)
```

```
    ax2.set_xlabel('Traveler Number', fontsize=12)
    ax2.set_ylabel('Wait Time (minutes)', fontsize=12)
    ax2.grid(True, alpha=0.3)
    ax2.set_ylim(0,420)


    plt.tight_layout()
    plt.show()
```

```
# Run the simulation and create plots
for num_queues in range(3, 11):
    num_simulations=3
    run_until=60*24 #1 DAYS
    run_and_plot_simulations(num_simulations, run_until, num_queues)
```

6.5 Appendix E - Simulation of Key Performance Indicators

```
# Plotting the graphs for key performance indicators including
# average queue length, maximum queue length, and average waiting times.

import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt
from tqdm import tqdm

def run_simulations(num_simulations, run_until, num_queues):

    # Store results
    all_queue_lengths = []
    all_max_lengths = []
    all_wait_times = []

    # Parameters
    arrival_rate = 10
    service_rate = 2

    # Service time distribution parameters
    mean_service = 1/service_rate
    sigma_service = 1/6
    a_service = (0 - mean_service) / sigma_service
    b_service = np.inf

    # Additional screening parameters
    mean_as = 2
    sigma_as = 2
    a_as = (0 - mean_as) / sigma_as
```

```python
    b_as = np.inf

    # Create distributions
    arrival_distribution = sts.expon(scale=1/arrival_rate)
    service_distribution = sts.truncnorm(a_service, b_service, loc=mean_service,
    scale=sigma_service)
    additional_screening_distribution = sts.truncnorm(a_as, b_as, loc=mean_as,
    scale=sigma_as)

    # Set up progress bar to track simulation
    progress_bar = tqdm(total=num_simulations, desc = f"{num_queues} Queues")

    for sim in range(num_simulations):
        # Create new airport instance for each simulation
        airport = Airport(arrival_distribution, service_distribution,
                          additional_screening_distribution, num_queues)

        # Run simulation
        queue_lengths, wait_times = airport.run(run_until)

        # Process key indicators into lists
        avg_lengths = [np.mean(lengths) for lengths in queue_lengths]
        max_lengths = [np.max(lengths) for lengths in queue_lengths]
        avg_waits = [np.mean(times) for times in wait_times]

        # Record the lists
        all_queue_lengths.extend(avg_lengths)
        all_max_lengths.extend(max_lengths)
        all_wait_times.extend(avg_waits)

        # Update tqdm
        progress_bar.update(1)

    return all_queue_lengths, all_max_lengths, all_wait_times

def calculate_population_ci(data, confidence=0.95):
    '''
    Calculate mean and population confidence interval.
    '''
    mean = np.mean(data)
    std = np.std(data)

    # For population interval, we use normal distribution quantiles
    z_score = sts.norm.ppf((1 + confidence) / 2)
    margin = z_score * std

```

```
    # Calculate interval that contains confidence% of the population
    ci_lower = mean - margin
    ci_upper = mean + margin

    return mean, (ci_lower, ci_upper)

def plot_results(queue_lengths, max_lengths, wait_times, num_queues):
    # Calculate confidence intervals and means
    ql_mean, ql_ci = calculate_population_ci(queue_lengths)
    ml_mean, ml_ci = calculate_population_ci(max_lengths)
    wt_mean, wt_ci = calculate_population_ci(wait_times)

    # Create figure with subplots
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

    # Plot average queue lengths
    ax1.hist(queue_lengths, bins=30, edgecolor='black', alpha=0.7)
    ax1.axvline(ql_mean, color='r', linestyle='--', label=f'Mean: {ql_mean:.2f}')
    ax1.axvline(ql_ci[0], color='g', linestyle=':', label=f'95% CI: [{ql_ci[0]:.2f}, {ql_ci[1]:.2f}]')
    ax1.axvline(ql_ci[1], color='g', linestyle=':')
    ax1.set_title(f'Average Queue Lengths with {num_queues} Queues in a Day')
    ax1.set_xlabel('Average Queue Length')
    ax1.set_ylabel('Frequency')
    ax1.legend()

    # Plot maximum queue lengths
    ax2.hist(max_lengths, bins=30, edgecolor='black', alpha=0.7)
    ax2.axvline(ml_mean, color='r', linestyle='--', label=f'Mean: {ml_mean:.2f}')
    ax2.axvline(ml_ci[0], color='g', linestyle=':', label=f'95% CI: [{ml_ci[0]:.2f}, {ml_ci[1]:.2f}]')
    ax2.axvline(ml_ci[1], color='g', linestyle=':')
    ax2.set_title(f'Maximum Queue Lengths with {num_queues} Queues in a Day')
    ax2.set_xlabel('Maximum Queue Length')
    ax2.set_ylabel('Frequency')
    ax2.legend()

    # Plot average waiting times
    ax3.hist(wait_times, bins=30, edgecolor='black', alpha=0.7)
    ax3.axvline(wt_mean, color='r', linestyle='--', label=f'Mean: {wt_mean:.2f}')
    ax3.axvline(wt_ci[0], color='g', linestyle=':', label=f'95% CI: [{wt_ci[0]:.2f}, {wt_ci[1]:.2f}]')
    ax3.axvline(wt_ci[1], color='g', linestyle=':')
    ax3.set_title(f'Average Waiting Times with {num_queues} Queues in a Day')
    ax3.set_xlabel('Average Wait Time (minutes)')
    ax3.set_ylabel('Frequency')
```

```python
    ax3.legend()

    plt.tight_layout()
    plt.show()

    # Return confidence intervals for summary statistics
    return (ql_mean, ql_ci), (ml_mean, ml_ci), (wt_mean, wt_ci)

```

```python
# For queue length = 7

# Run simulations (keeping your existing run_simulations function)
print("Starting simulations...")
num_simulations = 500
run_until = 60*24  # One day (60 mins * 24)
num_queues = 7
queue_lengths, max_lengths, wait_times = run_simulations(num_simulations, run_until,
num_queues)

# Plot results and get confidence intervals
ql_stats, ml_stats, wt_stats = plot_results(queue_lengths, max_lengths, wait_times,
num_queues)

# Print detailed summary statistics
print("\nDetailed Summary Statistics:")
print("\nAverage Queue Length:")
print(f"  Mean: {ql_stats[0]:.2f}")
print(f"  95% CI: [{ql_stats[1][0]:.2f}, {ql_stats[1][1]:.2f}]")

print("\nMaximum Queue Length:")
print(f"  Mean: {ml_stats[0]:.2f}")
print(f"  95% CI: [{ml_stats[1][0]:.2f}, {ml_stats[1][1]:.2f}]")

print("\nAverage Waiting Time (minutes):")
print(f"  Mean: {wt_stats[0]:.2f}")
print(f"  95% CI: [{wt_stats[1][0]:.2f}, {wt_stats[1][1]:.2f}]")
```

```python
# For queue length = 10

# Run simulations (keeping your existing run_simulations function)
print("Starting simulations...")
num_simulations = 500
run_until = 60*24  # One day (60 mins * 24)
num_queues = 10
queue_lengths, max_lengths, wait_times = run_simulations(num_simulations, run_until,
num_queues)
```

```python

# Plot results and get confidence intervals
ql_stats, ml_stats, wt_stats = plot_results(queue_lengths, max_lengths, wait_times,
num_queues)

# Print detailed summary statistics
print("\nDetailed Summary Statistics:")
print("\nAverage Queue Length:")
print(f"  Mean: {ql_stats[0]:.2f}")
print(f"  95% CI: [{ql_stats[1][0]:.2f}, {ql_stats[1][1]:.2f}]")

print("\nMaximum Queue Length:")
print(f"  Mean: {ml_stats[0]:.2f}")
print(f"  95% CI: [{ml_stats[1][0]:.2f}, {ml_stats[1][1]:.2f}]")

print("\nAverage Waiting Time (minutes):")
print(f"  Mean: {wt_stats[0]:.2f}")
print(f"  95% CI: [{wt_stats[1][0]:.2f}, {wt_stats[1][1]:.2f}]")
```