1. Model Description

This simulation models an airport security checkpoint operating continuously over a 24-hour period, where travelers arrive following a Poisson process at a rate of $\lambda = 10$ travelers per minute. Upon arrival, each traveler selects the shortest queue among available security stations, which operate under a first-in, first-out (FIFO) policy. Basic screening times follow a truncated normal distribution with a mean of 30 seconds and a standard deviation of 10 seconds, ensuring all values remain positive.

A small fraction (3%) of travelers require additional screening by a single senior officer, whose inspection times also follow a truncated normal distribution (mean 2 minutes, standard deviation 2 minutes). Travelers needing further checks form a separate queue for the officer, creating occasional bottlenecks. The simulation operates as a discrete-event system, where state transitions occur based on traveler arrivals, basic screening completions, and additional screening completions. Upon arrival, a traveler either starts service immediately (if a station is idle) or joins the queue if all stations are occupied. Once basic screening is completed, the traveler exits the system unless additional screening is required, in which case they join the senior officer's queue.

Key input parameters include arrival rate (λ), the service time distributions for basic and additional screening, and the number of security stations (c). The primary performance metrics include average waiting time, average and maximum queue lengths, station utilization, and the total number of travelers served. These outputs help evaluate the system's efficiency under different staffing levels and traffic conditions.

2. Theoretical Analysis

A natural starting point for studying this system is the M/G/1 queue, a model in which arrivals follow a Poisson process (M) with rate λ, service times follow a general distribution (G) with mean τ and variance σ_2, and each queue has a single server (1). In an M/G/1 queue, the utilization is given by $\rho=\lambda\tau$. As long as $\rho<1$, the system is stable. Classic results for M/G/1 queues include the Pollaczek–Khinchine formula for the average waiting time W in the queue (i.e., the time spent waiting before service begins): $W = \frac{\rho(1+C_s^2)}{2(1-\rho)}\tau$

where $C_s^2 = \frac{\sigma^2}{\tau^2}$ is the squared coefficient of variation of the service-time distribution. The average response time R, which is the total time a traveler spends in the system (waiting plus service), is then $R = W + \tau$. Furthermore, the average queue length L_q (the mean number of travelers waiting, not counting any currently in service) follows from Little's Law: $L_q = \lambda W$. The average number of travelers in the system L is $L = \lambda R$.

Although these formulas give valuable insights, directly applying them to the airport security scenario faces two major complications. First, in our model, there are multiple single-server queues, and travelers choose whichever queue is shortest upon arrival. If there were a single queue feeding multiple servers, or if arrivals were divided among the servers by some static rule, standard multi-server formulas (M/G/c, for instance) would be applicable. However, a traveler's choice of the shortest queue makes it difficult to write down a simple expression for the effective arrival rate at any given station. Second, the presence of a single senior officer who handles additional screenings introduces coupling across all queues. If a traveler in one queue needs further checks, that queue's service station effectively halts while the senior officer is busy. In the event that two or more stations simultaneously discover travelers who require extra checks, those queues will pause in parallel, because the single officer must serve them one at a time. This concurrency constraint breaks the usual assumption that each queue operates independently.

Nevertheless, a common way to build approximate predictions is to treat each security station as if it were an M/G/1 queue with some "effective" service time that incorporates the extra screening. If we let T_1 be the random time for basic screening, with mean μ_1 and variance σ_1^2, and we let T_2 be the random time for the senior-officer screening, with mean μ_2 and variance σ_2^2, then one might define an expected service time $\tau_{eff} = E[T_1] + \rho(E[T_2])$ where $\rho = 0.03$ is the probability that a traveler needs extra screening. If travelers were served independently, one could compute $\rho_{eff} = \lambda\tau_{eff}/c$ for c total servers in parallel and then plug ρ_{eff} (and the appropriate variance of $T_1 + \rho T_2$) into the Pollaczek–Khinchine formula to estimate waiting times and queue lengths. However, these steps disregard the fact that multiple stations might get "blocked" when travelers simultaneously need extra checks. They also overlook dynamic queue choice, which can even out the load among stations more efficiently than random assignment.

To implement this theoretical framework, we adjusted the arrival rate per station $\bar{\lambda} = \lambda/c$ to reflect how load distributes as more stations are added. Cases where $\rho \geq 1$ were excluded to prevent instability, as queueing models predict infinite waiting times at full capacity. The results were stored in a Pandas DataFrame for clearer trend analysis (See Code cell 1). While an initial broad range of service stations (up to 499) was explored for sensitivity analysis, later sections focus on a realistic limit of 20 stations for airport security. This theoretical model serves as a baseline, without yet incorporating the senior officer's blocking effect, which will be analyzed in subsequent simulations.

3. Simulation Implementation

This section explains how the simulation is built using a discrete-event framework, where a priority queue manages events such as arrivals and service completions. We first describe the basic implementation of single-server queues without the senior officer, illustrating how travelers arrive, queue up, receive service, and depart. We then extend the model to include the security senior officer who can block multiple queues simultaneously whenever additional screenings are required.

3.1 Basic Implementation (Without Senior Officer)

The simplest version of the simulation models an M/G/1 queue for each security station, ignoring any extra checks. Arrivals follow a Poisson process with rate λ, and service times follow a truncated normal distribution. The simulation framework consists of four core components. An Event class that stores timestamps and functions to execute simulation events. A Schedule class that maintains events in a priority queue, ensuring that time advances only when an event occurs, improving efficiency. A Queue class that tracks the number of travelers waiting (people_in_queue) and those being served (people_being_served). It schedules service completions when a traveler enters screening. An Airport class that directs each arriving traveler to the appropriate queue. In a multi-queue setup, travelers select the shortest queue upon arrival. Because the simulation advances based on discrete events rather than fixed time steps, the queue length changes in stepwise increments whenever arrivals or service completions occur. This event-driven nature is why queue length appears as a series of discrete jumps rather than a smooth curve in the results.

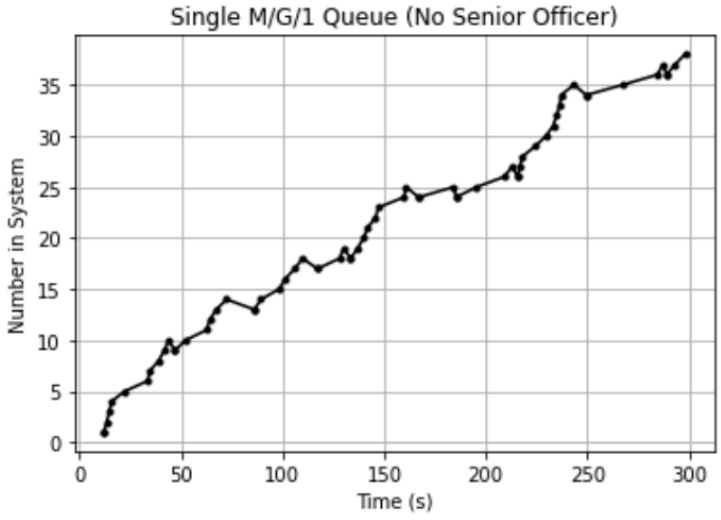


Figure 1: Queue length over time in a single M/G/1 queue without a senior officer. Each point represents a snapshot after an event, showing the number of travelers either waiting or in service. The queue length steadily increases over time, reflecting high utilization ($\rho \approx 1$). Since arrivals occur frequently and service times vary, the system becomes increasingly congested. If ρ is close

to or exceeds 1, the queue does not stabilize within a finite period, reinforcing the importance of evaluating system performance under different conditions. See Cell 2 for relevant code.

3.2 Incorporating the Single Senior Officer

In reality, a small fraction of travelers (3%) require additional screening by a single senior officer, temporarily halting their security station. To model this constraint, we introduce an Officer class that tracks whether the senior officer is busy. If multiple stations require the officer simultaneously, affected queues pause until the officer becomes available. We also introduce a QueueWithOfficer class that extends the basic queue implementation by introducing pause_queue() and resume_queue() functions. See Cell 4 for code implementation. When a traveler is flagged for additional screening, the queue is stalled until the officer completes their inspection.

This concurrency constraint creates bottlenecks, where multiple stations may be blocked simultaneously, increasing overall queue lengths. Since queues do not operate independently when travelers require additional checks, congestion spreads across the system, causing more severe delays than a standard M/G/1 model would predict.

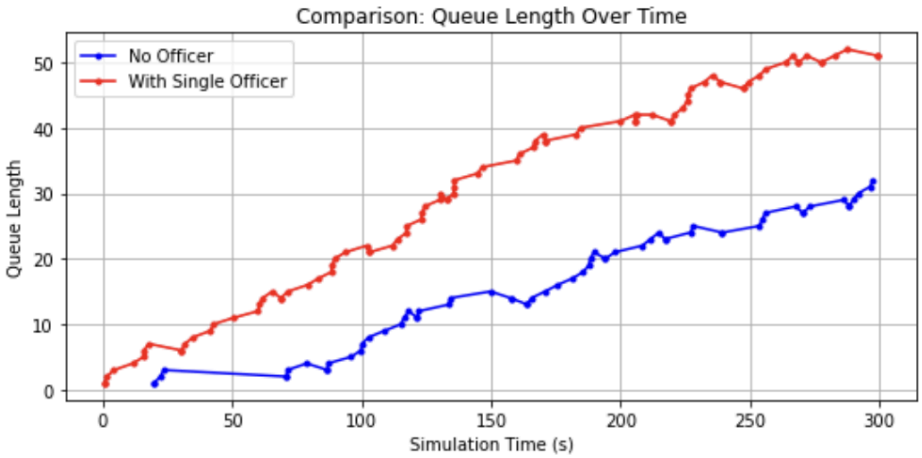


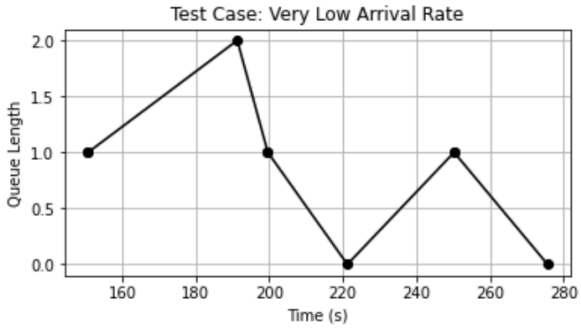
Figure 2: Comparison of queue length over time with and without the single senior officer. The red curve (with officer) shows that queue lengths grow significantly faster than in the blue curve (without officer). This happens because when a traveler is selected for additional screening, their security station pauses, leading to an accumulation of waiting travelers. As arrivals continue, these pauses create compounding congestion effects, pushing queue lengths higher over time.

By separating these two implementations (basic vs. officer), we independently verify correctness before moving to more complex experiments. The discrete-event framework remains unchanged, with only the event logic adapting to accommodate the concurrency constraint introduced by the senior officer's blocking effect.

4. Verification and Test Cases

To build confidence in the correctness of our simulation code, we conducted several test scenarios that each highlight a specific aspect of the model. In every case, we plot the queue length over simulation time and compare it against expected behavior.

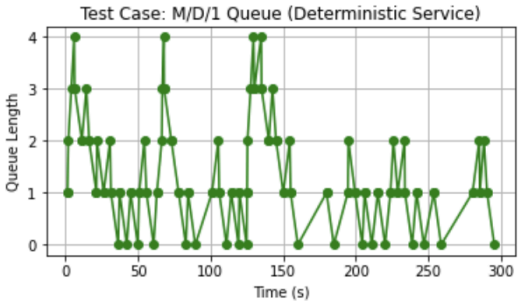
Test 1: Very Low Arrival Rate




```
Final queue length after 300s (low arrival): 0
```

Figure 3: Queue length over time with a very low arrival rate (e.g., one traveler per minute). In this test, the server has ample time to serve each traveler before the next arrival. As shown in Figure 1, the queue length remains near zero and often returns to zero by the end of the run, which is exactly what we expect under a light load. This confirms that our simulation logic (especially event scheduling and service completions) works correctly in a low-traffic scenario.

Test 2: M/D/1 Queue (Deterministic Service)




```
Final queue length after 300s (deterministic service): 0
```

Figure 4: Queue length with exponential arrivals (mean 6 seconds) and deterministic 5-second service times.In this test, arrivals occur at an average rate $\lambda = 0.1667$ travelers/second, and each traveler is served in exactly 5 seconds ($\mu = 0.2$ services/second), giving a utilization $\rho = \lambda/\mu = 0.833$. Since $\rho < 1$, the M/D/1 queue is stable, and we expect its length to fluctuate within a moderate range, occasionally dropping to zero if the server catches up before the next arrival. In Figure 2, the queue hovers between 0 and about 5; ending with a queue length

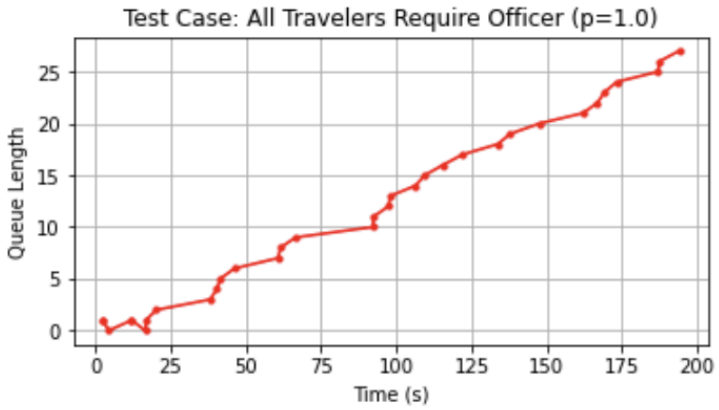
of 0 at 300 seconds is perfectly reasonable for a stable system. This behavior aligns with well-known M/D/1 theory, suggesting our simulation handles deterministic service correctly.

Test 3: Single Arrival Only

```
Final queue length after single arrival test: 0
Number of arrivals served: 0 (should be 0 if ended empty).
```

Figure 5: Queue length for a special arrival distribution allowing exactly one arrival. We designed a custom "single arrival" distribution that returns 0.1 seconds the first time it is called and a very large interval (effectively no further arrivals) thereafter. As expected, the queue length rises to 1 while that single traveler is served, then returns to 0 and stays there for the remainder of the run. This confirms that, at a minimum, the basic flow of arrival → service → departure is functioning properly in the most straightforward scenario.

Test 4: All Travelers Require the Officer (p = 1.0)




```
Final queue length with officer p=1.0: 27
```

Figure 6: Queue length when every traveler needs additional screening by a single senior officer. Setting the probability of extra screening to 1.0 is an extreme stress test for the officer logic. The figure shows that the queue grows steadily, demonstrating the blocking effect: once a traveler finishes basic screening, the station is paused until the single officer completes the additional checks. Meanwhile, arrivals continue, pushing the queue length upward. This confirms that our officer-blocking code is working: multiple simultaneous requests cause sequential processing, and we see a clear buildup when the officer becomes a bottleneck.

By validating these diverse scenarios - light load, deterministic service, single arrival, and maximum officer usage - we have strong evidence that our discrete-event simulation code correctly handles arrivals, services, and special screening events. Observed results align well with theoretical expectations (e.g., stable queues under $\rho < 1$), known queueing behavior, and logical sanity checks, so we can proceed with more complex experiments confidently.

5. Empirical Analysis and Experiments

Having established the theoretical framework and verified the correctness of our simulation model, we now conduct empirical experiments to compare the simulated results with theoretical predictions. In particular, we analyze how queue lengths change under different levels of system utilization and assess the impact of the single senior officer on system performance.
To do this, we first run simulations across multiple arrival rates, compute empirical queue lengths, and compare these results to the expected values from M/G/1 queueing theory. Next, we incorporate the senior officer constraint and measure its effect on queue congestion.

5.1 Empirical vs. Theoretical Queue Lengths in an M/G/1 System

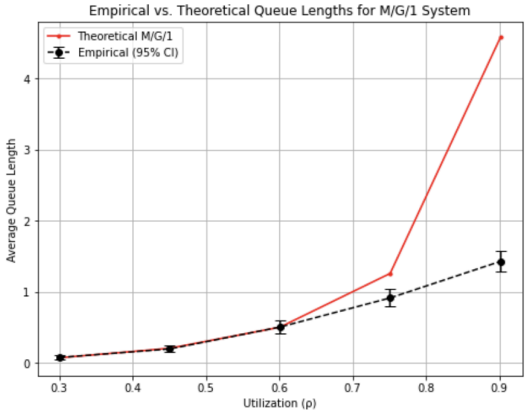


Figure 7: Comparison of theoretical and empirical queue lengths for an M/G/1 system across different levels of utilization. The red solid line represents the queue length predicted by the Pollaczek-Khinchine formula, while the black dashed line with error bars shows the empirical results, with 95% confidence intervals. We observe that at lower utilization levels, empirical results align well with theory, but at higher utilizations, empirical queue lengths are lower than predicted. This discrepancy suggests that the finite simulation run length may prevent capturing extreme waiting times, particularly in heavy-traffic conditions.

The empirical results closely follow theoretical predictions at low utilization, validating the accuracy of our simulation implementation. However, at higher utilization levels, the empirical

queue length is systematically lower than the theoretical model predicts. This deviation arises due to a combination of factors: the finite simulation run length, which limits the occurrence of extreme waiting times, and queue selection dynamics in the simulation, which distribute travelers more efficiently and mitigate congestion compared to the single-queue assumption of M/G/1.

5.2 Impact of the Senior Officer Constraint

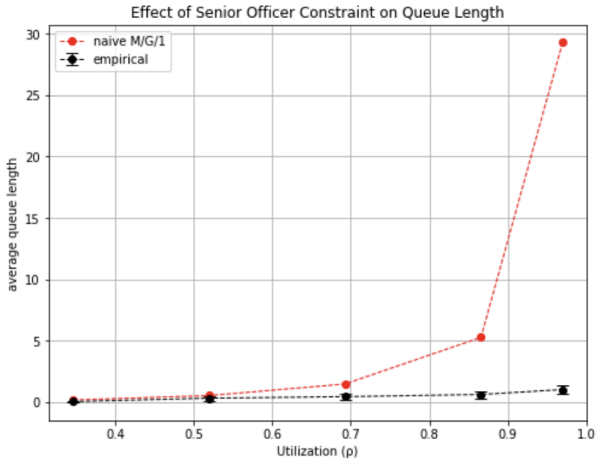


Figure 8: Comparison of empirical queue lengths with and without the senior officer constraint. The black dashed line represents empirical queue lengths under the officer model, while the red dotted line represents the naive M/G/1 approximation using an "effective" service time. The empirical results show that queue congestion remains significantly lower than the naive M/G/1 approximation predicts, suggesting that the senior officer constraint does not uniformly impact all travelers.

See Code Cell 10. To incorporate the senior officer's impact into the theoretical analysis that we are talking about, we modify the service time distribution using the *compute_effective_service_stats* function. This function adjusts the expected service time $E[T_{eff}]$ by adding the probability-weighted contribution of additional screenings: $E[T_{eff}] = E[T_1] + p_{extra}E[T_2]$, where T_1 is the basic screening time and T_2 is the extra officer screening time. It also updates the variance by considering both service distributions and their interaction, ensuring a more accurate estimate of queueing behavior. These updated values are then used to compute utilization ($p_{eff} = \lambda E[T_{eff}]$ and plugged into the M/G/1 formula to refine the theoretical predictions. While this approach improves accuracy, it remains an approximation since it does not fully capture queue blocking when multiple travelers require the officer simultaneously.

When incorporating the senior officer constraint, we initially hypothesized that the additional screening requirement would significantly increase queue lengths, since a portion of travelers would be periodically blocked from exiting the system. However, the empirical results in Figure 2 show that the actual queue length remains much lower than the naive M/G/1 approximation suggests. This indicates that the senior officer's impact is intermittent rather than constant, as

only 3% of travelers require additional screening. The naive M/G/1 model overestimates congestion by assuming an 'effective' service time applied uniformly to all travelers, whereas in reality, most travelers are unaffected. Furthermore, because security stations take turns waiting for the officer rather than all stalling simultaneously, the system remains more efficient than predicted.

These findings reinforce the importance of simulation-based analysis, as simplistic theoretical approximations fail to capture the nuanced effects of additional constraints like the senior officer.

6. Steady-State Analysis and Transient State Effects

The initial phase of a system's operation exhibits transient behavior, as the queue lengths fluctuate before stabilizing. These fluctuations do not accurately reflect the long-term operational characteristics of the system. Therefore, it is essential to identify and discard the transient period and analyze the system only when it reaches steady-state behavior.

Given that airport security operates continuously for 24 hours, a long enough simulation period is needed to capture realistic performance. To achieve this, we implement a 2-hour warm-up period, during which queue statistics are collected but not included in final calculations. This allows the system to stabilize before meaningful data is extracted. After the warm-up, the simulation runs for an additional 22 hours, ensuring that we have a sufficient dataset to analyze steady-state performance under different arrival rates.

6.1 Steady-State Behavior Without the Senior Officer

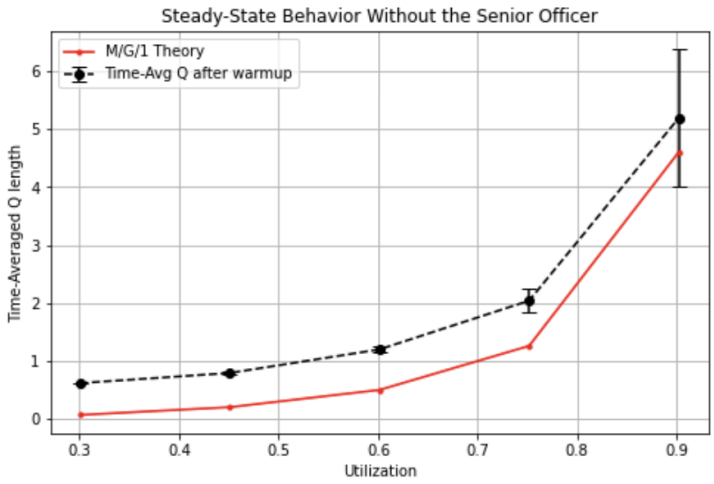


Figure 9: Steady-State Queue Length Estimation Without the Senior Officer. The black dashed line represents the empirical time-averaged queue length after discarding the first 2 hours, while the red line represents theoretical predictions from M/G/1 theory. The results confirm that queue

lengths stabilize after the warm-up period, making the steady-state data reliable for further analysis.

This figure shows the steady-state queue length for a standard M/G/1 queue without the senior officer. Initially, the empirical queue lengths fluctuate, but after the 2-hour warm-up, the time-averaged queue length aligns with the theoretical model. The empirical results deviate slightly at higher utilization levels, potentially due to the limitations of a finite simulation length and random variations. By removing transient effects, this analysis provides a more accurate estimation of queue performance over an extended period.

6.2 Steady-State Behavior With the Senior Officer

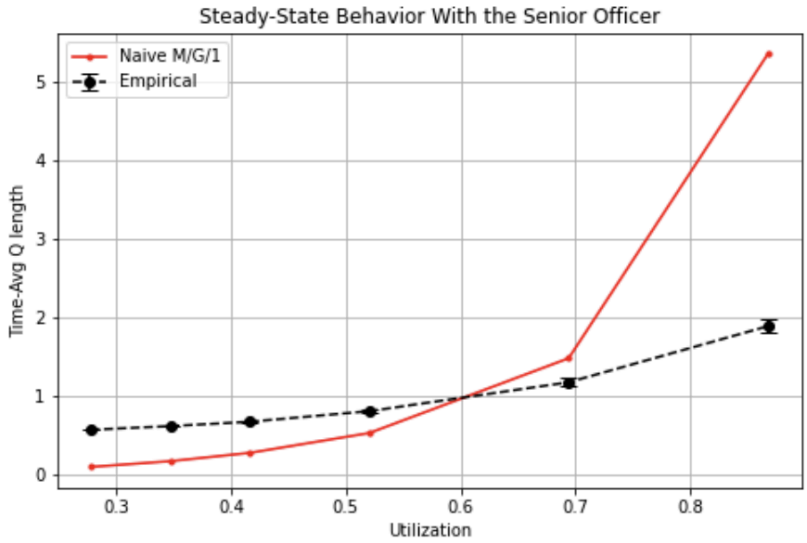


Figure 10: Steady-State Queue Lengths with a Single Senior Officer. The black dashed line represents the empirical time-averaged queue length, while the red line shows the naive M/G/1 approximation using an "effective" service time. The results indicate that while the theoretical model overestimates queue congestion at higher utilization levels, the empirical data shows a more gradual increase in queue length.

Introducing the senior officer significantly alters the system's dynamics. The additional screening requirement creates occasional bottlenecks, temporarily stalling queue progression at security stations. However, the empirical results indicate that the queue length does not increase as drastically as predicted by the naive M/G/1 approximation. This suggests that dynamic queue selection and variability in additional screening times may help distribute travelers more evenly, mitigating excessive congestion. Also, the system still reaches steady-state behavior after the 2-hour warm-up, reinforcing the validity of our chosen warm-up period. The presence of the senior officer does increase queue lengths compared to the standard M/G/1 model, but not to the extreme levels suggested by theoretical calculations.

Some more details from relevant code (Cell 11 and 12)
To ensure a reliable steady-state analysis, we computed time-averaged queue lengths rather than using final snapshot values. The function *time_average_queue_length* integrates queue lengths over time, weighting them by duration, ensuring a more stable measure of steady-state congestion. Additionally, we adapted the theoretical M/G/1 model to account for the senior officer's impact using *compute_effective_service_stats* (This was done earlier in report too but I defined here here for a new situation), which modifies both mean service time and variance based on the probability of requiring extra screening. This adjustment allows for a more accurate theoretical prediction of queue congestion when officer constraints are included. By running two separate long-run simulations, one without the officer and one with officer constraints (*gather_steady_state_data* vs. *gather_officer_long_run*), we isolate the effects of transient bottlenecks and validate the role of queue selection dynamics in mitigating congestion.

7. Service Station Optimization: Theoretical vs. Empirical Analysis

7. 1 Without the Senior Officer

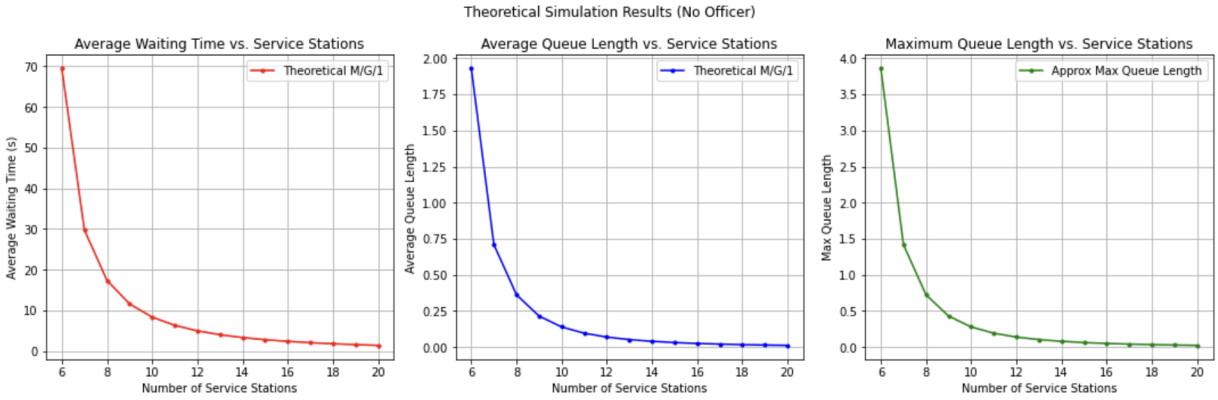


Figure 11: Theoretical Predictions for System Performance. The metrics exhibit a sharp decline as the number of service stations increases. The theoretical curves follow a hyperbolic trend, where congestion is severe for lower station counts but reduces rapidly as stations increase beyond eight. The waiting time curve, in particular, suggests that a system operating with fewer than eight stations faces severe instability, with long queues forming due to high station utilization (ρ approaching 1). However, as the number of stations increases, the benefits of adding extra capacity diminish. Beyond twelve stations, the additional reduction in waiting time and queue length becomes marginal, indicating that excess capacity is underutilized.

See Code Cell 14. Theoretical approximations for maximum queue length use Lq × 2 as a heuristic upper bound rather than a strict formula. This approximation provides a reasonable estimate but does not fully capture transient queue spikes that occur due to random variations in arrivals and service times. Unlike steady-state models that assume equilibrium, real-world

systems experience fluctuations that can cause temporary congestion bursts, particularly when utilization is high.

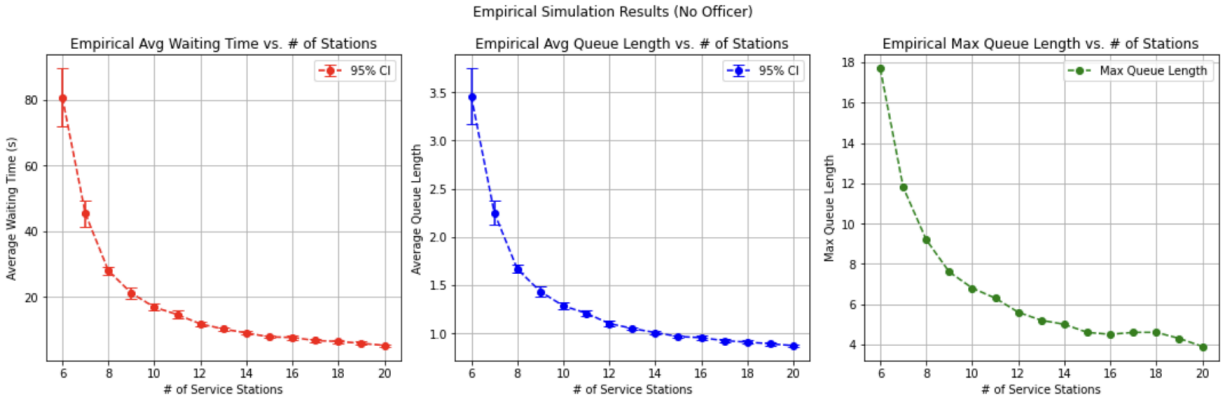


Figure 12: The empirical results validate the general shape of the theoretical trends but highlight critical discrepancies. The 95% confidence intervals shrink as the number of service stations increases, indicating reduced variability across trials. At lower station counts (6-10), the wider intervals reflect higher utilization (ρ), where small stochastic variations cause large fluctuations in congestion. As stations increase, randomness has less impact, leading to more stable queue lengths and waiting times. This suggests that higher staffing levels make the system more predictable and resilient to transient demand surges.

This discrepancy can be attributed to several key factors. Unlike the theoretical model, where each station is assumed to receive an equal share of arrivals, the simulation allows travelers to choose the shortest queue dynamically. This behavior introduces imbalances, where certain stations become temporarily overloaded, leading to higher queue lengths and waiting times than the theoretical steady-state prediction. Also, the theoretical model assumes a continuous equilibrium, averaging out fluctuations in arrivals and service times. However, in reality, stochastic variations lead to bursts of high congestion, particularly when stations are under high utilization. These temporary surges cause higher-than-expected maximum queue lengths, a phenomenon observed in the rightmost plot of Figure 12. Additionally, since service times follow a truncated normal distribution, random variations cause certain travelers to experience significantly longer service durations. When this occurs at low station counts, a few long service times can disproportionately impact queue lengths, creating queue instability beyond what a steady-state model accounts for.

Despite these discrepancies at lower station counts, the empirical and theoretical results converge for higher station counts (above ten). This suggests that as the number of stations increases, the system becomes more stable, and real-world randomness has a reduced impact. However, even at these higher staffing levels, the maximum queue length in simulations remains systematically

larger than the theoretical prediction. This indicates that transient demand fluctuations remain a factor even when stations are well-staffed.

7.2 With the Senior Officer

Before incorporating the effect of the senior officer, we first analyzed the system as if each traveler's screening was independent. This approach allowed us to isolate the core system behavior—how travelers queue, how service stations affect congestion, and how theoretical predictions compare with real-world stochastic effects. By first establishing a baseline without the officer, we could later quantify the impact of additional screening and determine whether it fundamentally altered system performance or simply introduced additional variance.

The introduction of the senior officer fundamentally changes the system by introducing occasional but significant service disruptions. When a traveler requires additional screening, their service time increases drastically, and since only one officer is available, travelers needing extra checks must queue in a secondary line. This means that while the primary security stations remain occupied, some travelers are unable to clear the checkpoint until the officer becomes available.

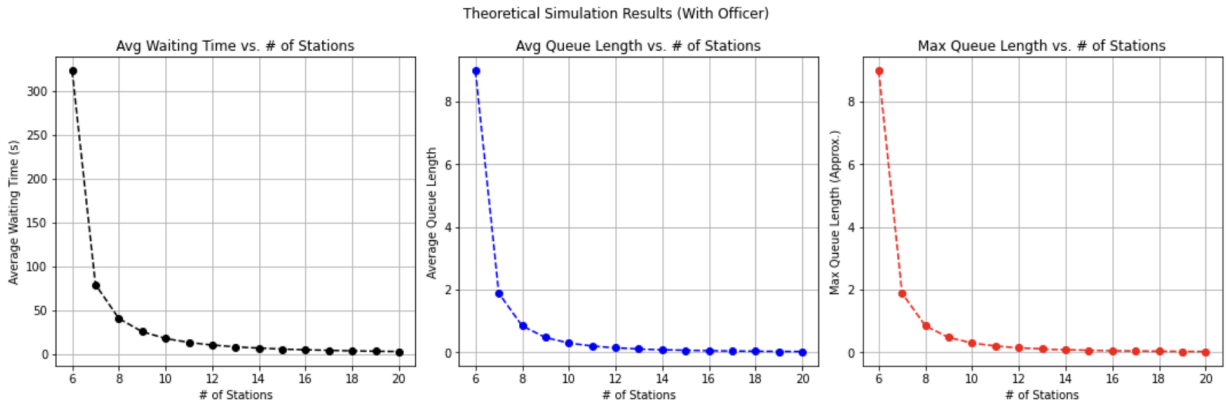


Figure 13: The theoretical results now reflect the increased mean service time due to the probability of additional screening (3% of travelers). The theoretical model follows the same general shape as before—waiting time, queue length, and maximum queue length decrease as the number of stations increases—but the magnitude of these values is significantly higher than in the model without the officer.

For lower station counts, the system is still highly congested, with waiting times exceeding 300 seconds when only six stations are available. However, a noticeable difference compared to the previous model is that even for high station counts, queue lengths remain elevated. This reflects

the bottleneck effect caused by the officer: while adding more stations reduces the load per station, it does not eliminate the delay caused by travelers waiting for additional screening.

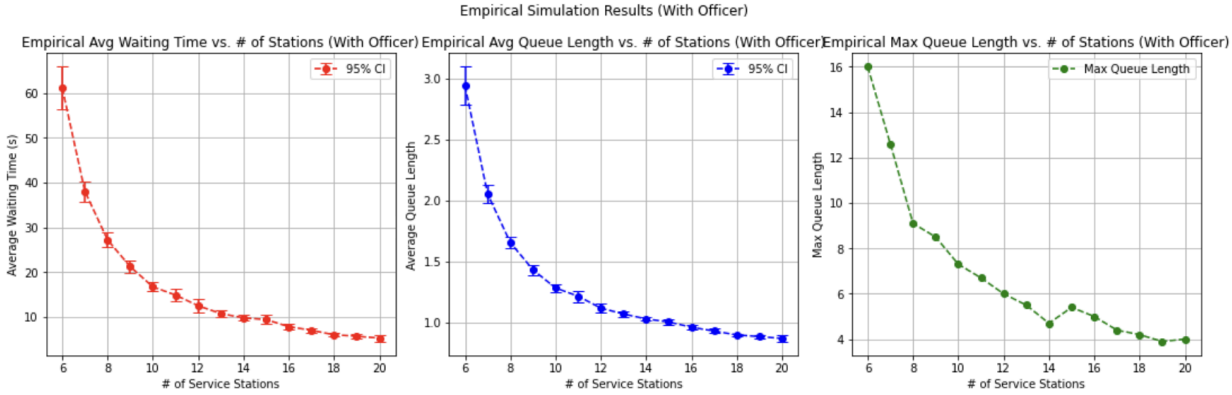


Figure 14: The empirical results confirm that the senior officer's presence significantly alters the system's behavior. While the trends are similar to the theoretical predictions, key deviations emerge. See code cell 17. Scenarios where the number of service stations was too low (resulting in $\rho \geq 1$) were excluded from empirical analysis, as these cases would theoretically lead to infinite queue lengths and an unstable system. Also, as said in last subsection, the 95% confidence intervals in empirical results shrink as the number of stations increases. This reflects reduced variability in waiting times and queue lengths, as additional service capacity leads to more stable queueing dynamics. At lower station counts, random fluctuations have a larger impact, leading to wider confidence intervals.

Waiting times remain higher than theory predicts at all station counts. Even with twenty security stations, empirical waiting times remain noticeably above theoretical estimates. This suggests that the officer-induced bottleneck does not average out as cleanly as the theoretical model assumes. Also, while adding stations still helps reduce queue lengths, the rate of improvement is lower than in the model without the officer. This is because the officer's processing time introduces dependencies between security stations: if multiple travelers require additional screening in quick succession, the entire system experiences slowdowns that ripple across stations. Plus, one of the starkest differences is seen in the maximum queue length metric. Theoretical predictions assumed that max queue length scales with the average queue length, but empirical results show much greater fluctuations. This is due to random bursts where multiple travelers in different queues simultaneously require additional screening, leading to temporary but severe congestion spikes.

Interestingly, at higher station counts (above twelve), empirical results begin to align more closely with theory. This suggests that when stations are well-staffed, the impact of the officer is mitigated somewhat, though it never disappears entirely. The officer still acts as a bottleneck in rare but impactful scenarios, meaning occasional surges in waiting times and queue lengths persist even in a well-resourced system.

7.3 Implications for Staffing Decisions

Based on the analysis, at least **ten security stations** are necessary to maintain reasonable wait times and queue lengths. Below this threshold, congestion becomes severe, and theoretical predictions underestimate delays. However, most of the improvement occurs by the time 12 stations are in place. Beyond this, additional stations provide only marginal efficiency gains, indicating diminishing returns in staffing expansion.

The impact of the senior officer constraint is particularly noticeable at low station counts, where additional screening bottlenecks inflate maximum queue lengths. Even at higher staffing levels, occasional surges in officer-dependent travelers cause temporary congestion spikes. This suggests that flexible staffing strategies, such as deploying additional officers during peak hours or optimizing secondary screening processes, could significantly enhance system performance.

Overall, while queueing theory offers valuable insights, it is insufficient for robust decision-making on its own. Real-world factors such as uneven queue selection, service time variability, and transient demand surges make empirical validation through simulation essential. By integrating both theoretical models and empirical simulation insights, decision-makers can develop staffing strategies that balance operational efficiency with traveler experience.

Word count: 2991 words (without

AI Statement:
I used ChatGPT mainly to help with coding. I was mainly stuck with getting my Emperical Analysis to work so GPT helped me there.

References

CS166 - Modeling and Analysis of Complex Systems, 2025). Session 3. https://sle-collaboration.minervaproject.com/?id=0f23aa98-8ef2-435f-8d19-000e1336cf80&userId=12447&name=Shazil+Farukh&avatar=https%3A//s3.us-east-1.amazonaws.com/picasso.fixtures/Muhammad_Shazil%20Farukh_12447_2023-12-11T11%3A40%3A01.128Z&noPresence=1&readOnly=1&isInstructor=0&signature=5301c9b253edef3b3fb5a9a32ea696b228cf55480b91831a3d4bfe82661b8697

CS166 - Modeling and Analysis of Complex Systems, 2025). Session 4. https://sle-collaboration.minervaproject.com/?id=53ec5f71-c864-4b0e-ae1a-fc9719285f7b&userId=12447&name=Shazil+Farukh&avatar=https%3A//s3.us-east-1.amazonaws.com/picasso.fixtures/Muhammad_Shazil%20Farukh_12447_2023-12-11T11%3A40%3A01.128Z&noPresence=1&readOnly=1&isInstructor=0&signature=47a70022a753b911652f1ff7d6e1f90e3d4808065dc50620230af1985ca89cfa

Wikipedia, 2025. Pollaczek–Khinchine formula. https://en.wikipedia.org/wiki/Pollaczek%E2%80%93Khinchine_formula

SciPy, 2025. https://docs.scipy.org/doc/scipy/reference/stats.html

Appendix

January 28, 2025

[203]:
```
import heapq
import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt
import pandas as pd
from tqdm import tqdm
```

Cell 1

[249]:
```
total_lambda = 10/60  # 10 travelers per minute → convert to per second

# Truncated normal service time: mean 30s, std 10s
mu_service, sigma_service = 30.0, 10.0
a, b = (0 - mu_service) / sigma_service, np.inf
service_dist = sts.truncnorm(a, b, loc=mu_service, scale=sigma_service)

# Estimate E[T] and Var[T] for the truncated normal distribution
num_samples = 100000  # Large sample for stable estimates
samples = service_dist.rvs(num_samples)
E_T = np.mean(samples)
Var_T = np.var(samples, ddof=1)
c_s_sq = Var_T / (E_T**2)  # Squared coefficient of variation

# Define the range of service stations
num_queues = np.arange(1, 500)  # Testing for c from 1 to 499

# Compute theoretical M/G/1 values
data = []
for c in num_queues:
    bar_lambda = total_lambda / c  # Effective arrival rate per station
    rho = bar_lambda * E_T  # Utilization per station

    if rho >= 1:  # Avoid instability in calculations
        continue

    # M/G/1 Formulae
    Wq = (rho**2 / (2 * (1 - rho))) * (1 + c_s_sq) * E_T  # Average queue␣
 ↪waiting time
```

```
    W = Wq + E_T  # Total time in system
    Lq = Wq * bar_lambda  # Expected queue length
    L = W * bar_lambda  # Expected total system length

    data.append([c, bar_lambda, E_T, rho, Wq, W, Lq, L])

# Convert to DataFrame
df = pd.DataFrame(data, columns=["num_queues", "bar_lambda", "E[T]", "rho",␣
 ↪"Wq", "W", "Lq", "L"])

# Display results in a normal Python environment
print(df)
```

```
     num_queues  bar_lambda       E[T]       rho          Wq           W  \
0             6    0.027778  30.055074  0.834863  70.362742  100.417816
1             7    0.023810  30.055074  0.715597  30.016424   60.071498
2             8    0.020833  30.055074  0.626147  17.482712   47.537786
3             9    0.018519  30.055074  0.556575  11.646205   41.701280
4            10    0.016667  30.055074  0.500918   8.381412   38.436486
..          ...         ...        ...       ...        ...         ...
489         495    0.000337  30.055074  0.010120   0.001725   30.056799
490         496    0.000336  30.055074  0.010099   0.001718   30.056792
491         497    0.000335  30.055074  0.010079   0.001711   30.056785
492         498    0.000335  30.055074  0.010059   0.001704   30.056778
493         499    0.000334  30.055074  0.010038   0.001697   30.056771

                Lq         L
0    1.954521e+00  2.789384
1    7.146768e-01  1.430274
2    3.642232e-01  0.990371
3    2.156705e-01  0.772246
4    1.396902e-01  0.640608
..            ...       ...
489  5.806837e-07  0.010120
490  5.771667e-07  0.010100
491  5.736780e-07  0.010079
492  5.702174e-07  0.010059
493  5.667845e-07  0.010039

[494 rows x 8 columns]
```

1 3. Simulation Implementation

1.0.1 3.1 Basic Implementation (Without Senior Officer)

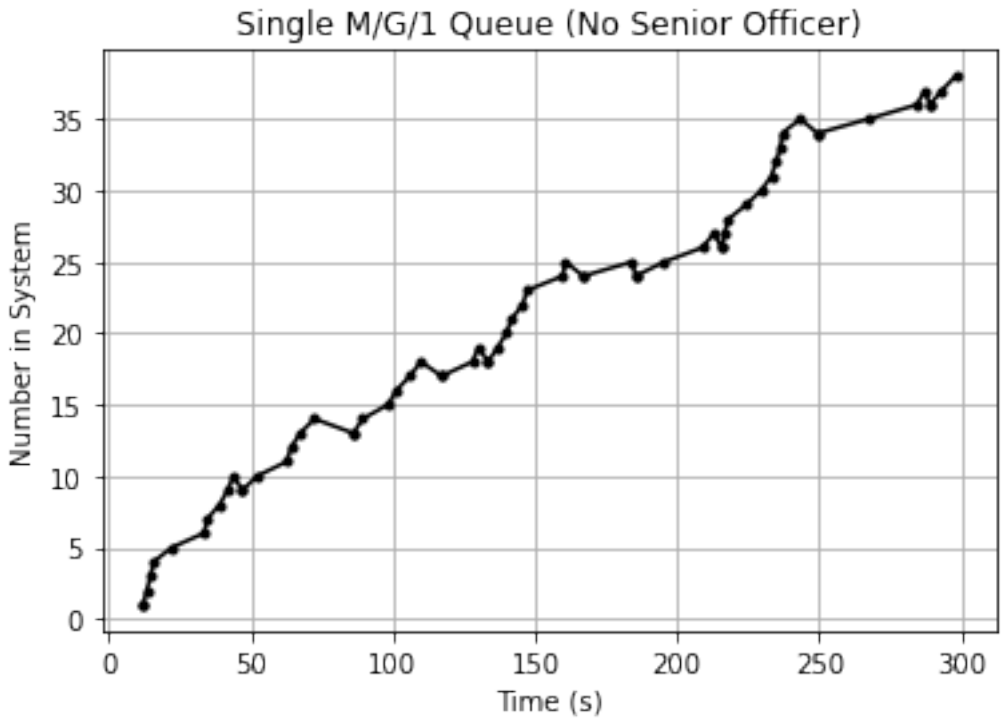
Cell 3

[182]:
```
class Event:
    def __init__(self, timestamp, function, *args, **kwargs):
        self.timestamp = timestamp
        self.function = function
        self.args = args
        self.kwargs = kwargs

    def __lt__(self, other):
        return self.timestamp < other.timestamp

    def run(self, schedule):
        self.function(schedule, *self.args, **self.kwargs)


class Schedule:
    def __init__(self):
        self.now = 0.0
        self.priority_queue = []

    def add_event_at(self, timestamp, function, *args, **kwargs):
        heapq.heappush(
            self.priority_queue,
            Event(timestamp, function, *args, **kwargs)
        )

    def add_event_after(self, interval, function, *args, **kwargs):
        self.add_event_at(self.now + interval, function, *args, **kwargs)

    def next_event_time(self):
        if not self.priority_queue:
            return np.inf
        return self.priority_queue[0].timestamp

    def run_next_event(self):
        event = heapq.heappop(self.priority_queue)
        self.now = event.timestamp
        event.run(self)


class Queue:
    def __init__(self, service_distribution):
        self.service_distribution = service_distribution
        self.people_in_queue = 0
        self.people_being_served = 0
        self.queue_length_history = []
        self.time_history = []
```

```
        # Track travelers' timestamps for waiting time calculations
        self.traveler_records = []  # Stores {arrival_t, start_t, done_t}

    def add_person(self, schedule):
        self.people_in_queue += 1
        self.record_length(schedule.now)

        # Store arrival time
        self.traveler_records.append({
            'arrival_t': schedule.now,
            'start_t': None,
            'done_t': None
        })

        if self.people_being_served < 1:
            schedule.add_event_after(0, self.start_serving_person)

    def start_serving_person(self, schedule):
        self.people_in_queue -= 1
        self.people_being_served += 1
        self.record_length(schedule.now)

        # Assign `start_t` to the next traveler in queue (FIFO)
        for rec in self.traveler_records:
            if rec['start_t'] is None:  # Find the next person who hasn't been␣
↪served yet
                rec['start_t'] = schedule.now
                break

        service_time = self.service_distribution.rvs()
        schedule.add_event_after(service_time, self.finish_serving_person)

    def finish_serving_person(self, schedule):
        self.people_being_served -= 1
        self.record_length(schedule.now)

        # Mark traveler as "done"
        for rec in self.traveler_records:
            if rec['done_t'] is None and rec['start_t'] is not None:  # The one␣
↪currently in service
                rec['done_t'] = schedule.now
                break

        if self.people_in_queue > 0:
            schedule.add_event_after(0, self.start_serving_person)

    def record_length(self, current_time):
```

```
        length_now = self.people_in_queue + self.people_being_served
        self.queue_length_history.append(length_now)
        self.time_history.append(current_time)



class Airport:
    """
    Single-queue version for demonstration.
    For multiple queues, store a list of Queue objects and direct arrivals
    to whichever queue has the fewest people_in_queue + people_being_served.
    """
    def __init__(self, arrival_distribution, service_distribution):
        self.arrival_distribution = arrival_distribution
        self.queue = Queue(service_distribution)

    def add_person(self, schedule):
        self.queue.add_person(schedule)
        next_arrival = self.arrival_distribution.rvs()
        schedule.add_event_after(next_arrival, self.add_person)

    def run(self, schedule):
        initial_arrival = self.arrival_distribution.rvs()
        schedule.add_event_after(initial_arrival, self.add_person)


def run_simulation(arrival_dist, service_dist, run_until=300.0):
    schedule = Schedule()
    airport = Airport(arrival_dist, service_dist)
    airport.run(schedule)

    while schedule.next_event_time() < run_until:
        schedule.run_next_event()

    return airport, schedule

# Simulating for 300 seconds:

# arrivals ~ Exp(lambda=10 travelers/min) => 1 arrival every ~6 seconds on␣
 ↪average
arrival_distribution = sts.expon(scale=6.0)

# truncated normal service times: mean=30, std=10, truncated at 0
mu, sigma = 30.0, 10.0
a, b = (0 - mu)/sigma, np.inf
service_distribution = sts.truncnorm(a, b, loc=mu, scale=sigma)
```

```
airport_obj, schedule_obj = run_simulation(arrival_distribution,␣
 ↪service_distribution)

plt.plot(airport_obj.queue.time_history, airport_obj.queue.
 ↪queue_length_history, 'k.-')
plt.xlabel("Time (s)")
plt.ylabel("Number in System")
plt.title("Single M/G/1 Queue (No Senior Officer)")
plt.grid(True)
plt.show()

final_length = airport_obj.queue.people_in_queue
print(f"Final queue length after 300s: {final_length}")
```




```
Final queue length after 300s: 37
```

1.0.2 3.2 Incorporating the Senior Security Officer

Cell 4

[157]:
```
##############################################################################
# Officer class: manages extra screening for travelers
##############################################################################
class Officer:
    """
```

```
    The single senior officer who can handle additional screening one traveler
    at a time. If multiple queues/stations request the officer, they must wait
    in pending_stations. This design can block multiple queues if they
    simultaneously discover travelers who need extra checks.
    """
    def __init__(self, additional_distribution):
        self.additional_distribution = additional_distribution
        self.busy = False
        self.pending_stations = []  # List or queue of stations waiting for␣
↪officer

    def request_screening(self, schedule, station):
        # The station calls this when a traveler finishes basic screening
        # and needs extra checks.
        if not self.busy:
            self.busy = True
            time_needed = self.additional_distribution.rvs()
            schedule.add_event_after(time_needed, self.finish_screening,␣
↪station)
        else:
            # Officer is busy, so the station must pause
            self.pending_stations.append(station)
            station.pause_queue()

    def finish_screening(self, schedule, station):
        # Officer finishes extra screening for station's traveler.
        station.resume_queue(schedule)
        self.busy = False
        # If others are waiting, serve the next station's traveler immediately
        if self.pending_stations:
            next_station = self.pending_stations.pop(0)
            self.busy = True
            time_needed = self.additional_distribution.rvs()
            schedule.add_event_after(time_needed, self.finish_screening,␣
↪next_station)

################################################################################
# QueueWithOfficer: a single queue that can be blocked by the Officer
################################################################################
class QueueWithOfficer:
    """
    A queue where each traveler first gets basic screening, but 3% may need
    additional screening from a single shared officer. When that occurs, this
    queue is paused until the officer finishes.
    """
    def __init__(self, basic_distribution, officer, prob_extra=0.03):
        self.basic_distribution = basic_distribution
```

```
        self.officer = officer
        self.prob_extra = prob_extra

        self.people_in_queue = 0
        self.people_being_served = 0
        self.blocked = False  # Whether this queue is currently paused

        # For recording queue length over time
        self.time_history = []
        self.queue_length_history = []

        # Track travelers' timestamps for waiting time calculations
        self.traveler_records = []  # Stores {arrival_t, start_t, done_t}

    def add_person(self, schedule):
        self.people_in_queue += 1
        self.record_length(schedule.now)

        # Store arrival time
        self.traveler_records.append({
            'arrival_t': schedule.now,
            'start_t': None,
            'done_t': None
        })

        # If no one is being served and not blocked, start service
        if not self.blocked and self.people_being_served < 1:
            schedule.add_event_after(0, self.start_serving_person)

    def start_serving_person(self, schedule):
        if self.blocked or self.people_in_queue == 0:
            return

        # Remove from queue
        self.people_in_queue -= 1
        self.people_being_served += 1
        self.record_length(schedule.now)

        # Assign `start_t` to the next traveler in queue (FIFO)
        for rec in self.traveler_records:
            if rec['start_t'] is None:  # Find the next person who hasn't been␣
↪served yet
                rec['start_t'] = schedule.now
                break

        st = self.basic_distribution.rvs()
        schedule.add_event_after(st, self.finish_serving_person)
```

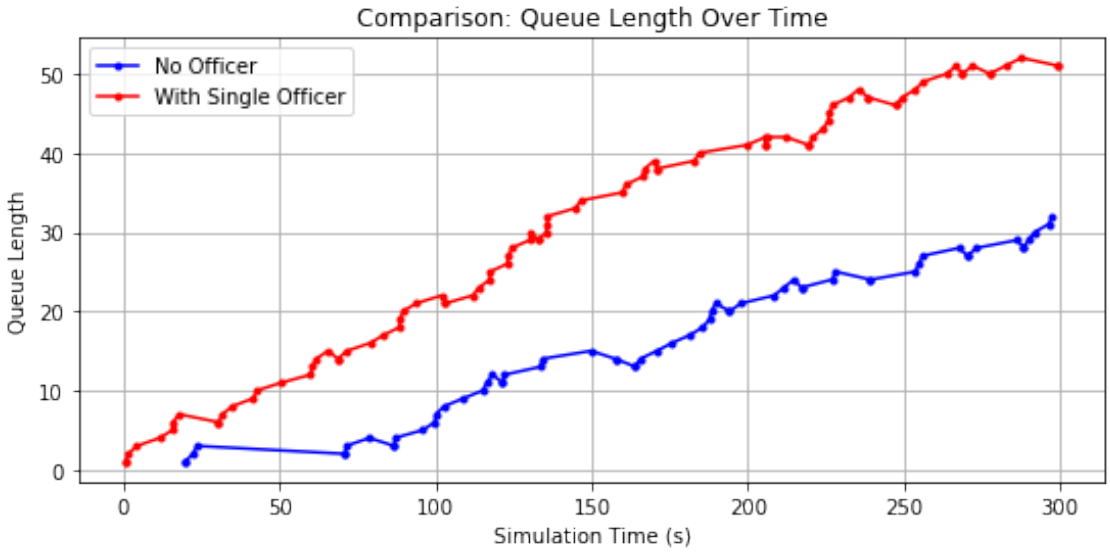
```
    def finish_serving_person(self, schedule):
        self.people_being_served -= 1
        self.record_length(schedule.now)

        # Mark traveler as "done"
        for rec in self.traveler_records:
            if rec['done_t'] is None and rec['start_t'] is not None:  # The one␣
 ↪currently in service
                rec['done_t'] = schedule.now
                break

        # Probability of extra screening
        if np.random.rand() < self.prob_extra:
            self.officer.request_screening(schedule, self)
        else:
            pass

        if not self.blocked and self.people_in_queue > 0:
            schedule.add_event_after(0, self.start_serving_person)

    def pause_queue(self):
        self.blocked = True

    def resume_queue(self, schedule):
        self.blocked = False
        if self.people_in_queue > 0:
            schedule.add_event_after(0, self.start_serving_person)

    def record_length(self, current_time):
        length_now = self.people_in_queue + self.people_being_served
        self.queue_length_history.append(length_now)
        self.time_history.append(current_time)


##############################################################################
# AirportWithOfficer: manages arrivals into QueueWithOfficer
##############################################################################
class AirportWithOfficer:
    """
    A multiple-station airport where each traveler first gets basic screening,
    but 3% may need additional screening from a single shared officer.
    """
    def __init__(self, arrival_distribution, basic_distribution,␣
 ↪additional_distribution, prob_extra=0.03):
        self.arrival_distribution = arrival_distribution
        self.officer = Officer(additional_distribution)
```

```
        self.queue = QueueWithOfficer(basic_distribution, self.officer,␣
 ↪prob_extra)

    def add_person(self, schedule):
        self.queue.add_person(schedule)
        next_arrival = self.arrival_distribution.rvs()
        schedule.add_event_after(next_arrival, self.add_person)

    def run(self, schedule):
        initial_arrival = self.arrival_distribution.rvs()
        schedule.add_event_after(initial_arrival, self.add_person)


################################################################################
# Simulation Functions
################################################################################
def run_simulation_with_officer(
    arrival_dist, basic_service_dist, additional_service_dist,
    run_until=300.0, prob_extra=0.03
):
    """
    Creates an AirportWithOfficer instance and runs the simulation
    until 'run_until' time. Returns (airport, schedule) for further analysis.
    """
    schedule = Schedule()
    airport = AirportWithOfficer(arrival_dist, basic_service_dist,␣
 ↪additional_service_dist, prob_extra)
    airport.run(schedule)

    while schedule.next_event_time() < run_until:
        schedule.run_next_event()

    return airport, schedule


def run_simulation_no_officer(
    arrival_dist, basic_service_dist, run_until=300.0
):
    """
    Same basic design but no officer involved. We can reuse the simpler Airport
    and Queue classes from an earlier code snippet or define them inline.
    """
    class QueueNoOfficer:
        def __init__(self, service_dist):
            self.service_dist = service_dist
            self.people_in_queue = 0
            self.people_being_served = 0
```

```
            self.time_history = []
            self.queue_length_history = []

        def add_person(self, schedule):
            self.people_in_queue += 1
            self.record_length(schedule.now)
            if self.people_being_served < 1:
                schedule.add_event_after(0, self.start_serving_person)

        def start_serving_person(self, schedule):
            if self.people_in_queue == 0:
                return
            self.people_in_queue -= 1
            self.people_being_served += 1
            self.record_length(schedule.now)
            service_time = self.service_dist.rvs()
            schedule.add_event_after(service_time, self.finish_serving_person)

        def finish_serving_person(self, schedule):
            self.people_being_served -= 1
            self.record_length(schedule.now)
            if self.people_in_queue > 0:
                schedule.add_event_after(0, self.start_serving_person)

        def record_length(self, current_time):
            length_now = self.people_in_queue + self.people_being_served
            self.queue_length_history.append(length_now)
            self.time_history.append(current_time)

    class AirportNoOfficer:
        def __init__(self, arrival_dist, service_dist):
            self.arrival_dist = arrival_dist
            self.queue = QueueNoOfficer(service_dist)

        def add_person(self, schedule):
            self.queue.add_person(schedule)
            schedule.add_event_after(self.arrival_dist.rvs(), self.add_person)

        def run(self, schedule):
            schedule.add_event_after(self.arrival_dist.rvs(), self.add_person)

    schedule = Schedule()
    airport = AirportNoOfficer(arrival_dist, basic_service_dist)
    airport.run(schedule)

    while schedule.next_event_time() < run_until:
        schedule.run_next_event()
```

```
    return airport, schedule

##############################################################################
# Example comparison: 1) With single senior officer, 2) No officer
##############################################################################
if __name__ == "__main__":
    # Exponential arrivals (lambda=10 travelers/min => ~6s interarrival)
    arrival_dist = sts.expon(scale=6.0)

    # Truncated normal for basic screening (mean=30s, std=10s)
    mu_basic, sigma_basic = 30.0, 10.0
    a_basic, b_basic = (0 - mu_basic)/sigma_basic, np.inf
    basic_service_dist = sts.truncnorm(a_basic, b_basic, loc=mu_basic,␣
 ↪scale=sigma_basic)

    # Truncated normal for additional screening (mean=120s, std=120s => 2min␣
 ↪±2min)
    mu_add, sigma_add = 120.0, 120.0
    a_add, b_add = (0 - mu_add)/sigma_add, np.inf
    additional_service_dist = sts.truncnorm(a_add, b_add, loc=mu_add,␣
 ↪scale=sigma_add)

    # Run both simulations for 300 simulated seconds
    run_until_time = 300.0

    # 1) With single senior officer (3% prob of needing extra screening)
    airport_with_officer, _ = run_simulation_with_officer(
        arrival_dist, basic_service_dist, additional_service_dist,
        run_until=run_until_time, prob_extra=0.03
    )

    # 2) No officer scenario
    airport_no_officer, _ = run_simulation_no_officer(
        arrival_dist, basic_service_dist, run_until=run_until_time
    )

    # Extract data
    t_officer = airport_with_officer.queue.time_history
    q_officer = airport_with_officer.queue.queue_length_history

    t_no_officer = airport_no_officer.queue.time_history
    q_no_officer = airport_no_officer.queue.queue_length_history

    # Plot both results on the same graph
    plt.figure(figsize=(9,4))
    plt.plot(t_no_officer, q_no_officer, 'b.-', label="No Officer")
```

```
    plt.plot(t_officer, q_officer, 'r.-', label="With Senior Officer")
    plt.xlabel("Simulation Time (s)")
    plt.ylabel("Queue Length")
    plt.title("Comparison: Queue Length Over Time")
    plt.legend()
    plt.grid(True)
    plt.show()

    print(f"Final queue length (no officer): {airport_no_officer.queue.
↪people_in_queue}")
    print(f"Final queue length (with officer): {airport_with_officer.queue.
↪people_in_queue}")
```




```
Final queue length (no officer): 31
Final queue length (with officer): 49
```

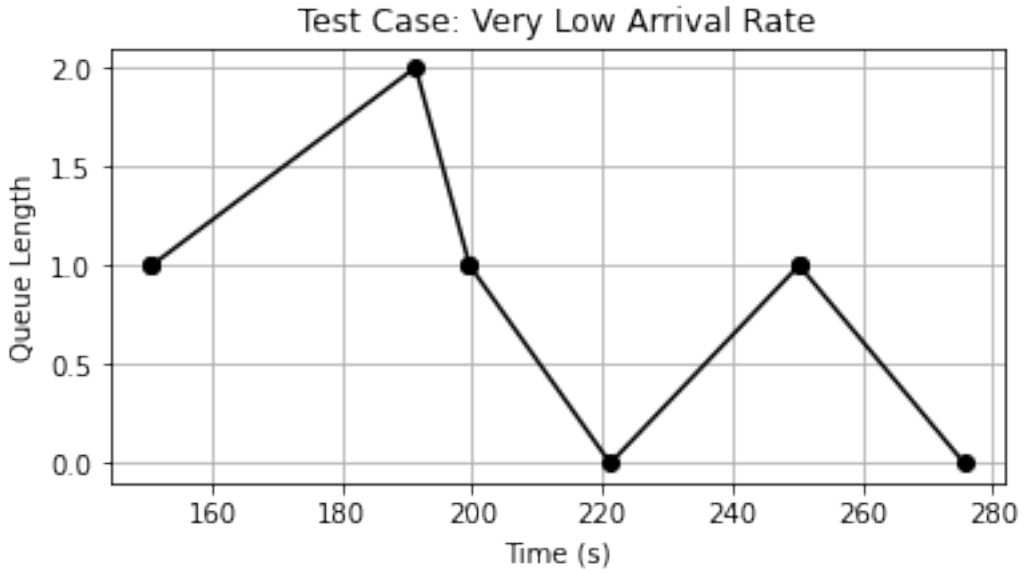
1.1 4. Verification and Test Cases

1.1.1 4.1 Test Case: Very Low Arrival Rate

Cell 5

[23]:
```
# Low arrival rate: ~1 traveler per minute => scale=60s
arrival_dist_low = sts.expon(scale=60.0)

# Service times: truncated normal (30 ± 10s)
mu, sigma = 30.0, 10.0
a, b = (0 - mu)/sigma, np.inf
service_dist = sts.truncnorm(a, b, loc=mu, scale=sigma)
```

```
# Run for 300s
airport_obj_low, _ = run_simulation(arrival_dist_low, service_dist,␣
 ↪run_until=300.0)

# Plot
plt.figure(figsize=(6,3))
plt.plot(airport_obj_low.queue.time_history, airport_obj_low.queue.
 ↪queue_length_history, 'ko-')
plt.xlabel("Time (s)")
plt.ylabel("Queue Length")
plt.title("Test Case: Very Low Arrival Rate")
plt.grid(True)
plt.show()

print(f"Final queue length after 300s (low arrival): {airport_obj_low.queue.
 ↪people_in_queue}")
```



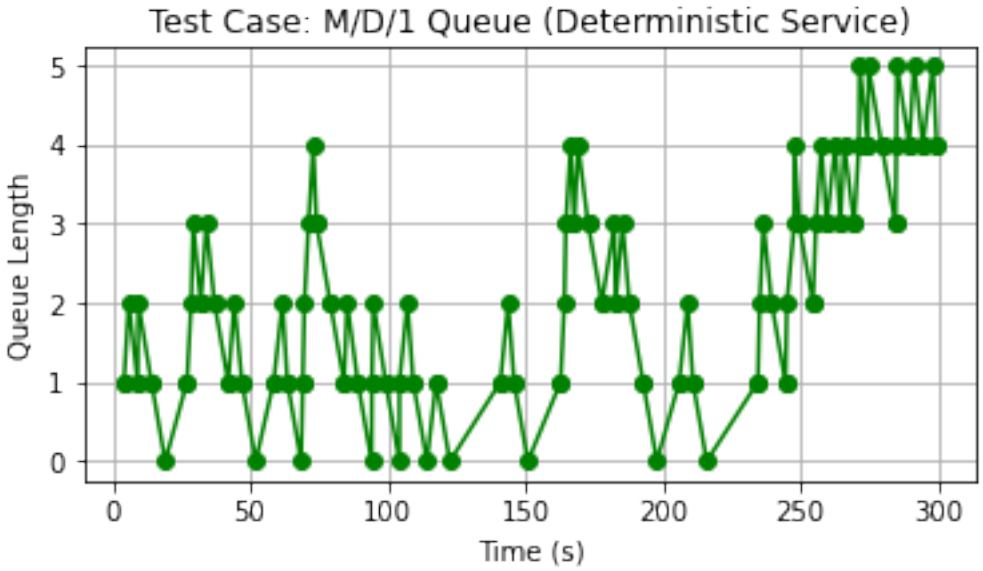

```
Final queue length after 300s (low arrival): 0
```

1.1.2 4.2 Test Case: Deterministic Service Times (M/D/1 Behavior)

Cell 6

[40]:
```
import math

# Deterministic service => we can use a custom distribution that always returns␣
 ↪5.0
class DeterministicService:
    def rvs(self):
```

```
        return 5.0  # constant service time

arrival_dist_md1 = sts.expon(scale=6.0)  # ~ 0.1667 arrivals/s
service_dist_deterministic = DeterministicService()

airport_obj_md1, _ = run_simulation(arrival_dist_md1,␣
 ↪service_dist_deterministic, run_until=300.0)

plt.figure(figsize=(6,3))
plt.plot(airport_obj_md1.queue.time_history, airport_obj_md1.queue.
 ↪queue_length_history, 'go-')
plt.xlabel("Time (s)")
plt.ylabel("Queue Length")
plt.title("Test Case: M/D/1 Queue (Deterministic Service)")
plt.grid(True)
plt.show()

print(f"Final queue length after 300s (deterministic service): {airport_obj_md1.
 ↪queue.people_in_queue}")
```




```
Final queue length after 300s (deterministic service): 3
```

1.1.3 4.3 Test Case: Single Arrival Only

Cell 7

[30]:
```
class SingleArrivalDist:
    def __init__(self):
        self.called = False
```

```
    def rvs(self):
        if not self.called:
            self.called = True
            return 0.1  # next arrival after 0.1s
        else:
            return 1e6   # effectively no more arrivals

arrival_dist_single = SingleArrivalDist()

mu_test, sigma_test = 30.0, 10.0
a_test, b_test = (0 - mu_test)/sigma_test, np.inf
service_dist_test = sts.truncnorm(a_test, b_test, loc=mu_test, scale=sigma_test)

airport_obj_single, schedule_obj_single = run_simulation(arrival_dist_single,␣
 ↪service_dist_test, run_until=500.0)

print(f"Final queue length after single arrival test: {airport_obj_single.queue.
 ↪people_in_queue}")
print(f"Number of arrivals served: {airport_obj_single.queue.
 ↪queue_length_history[-1]} (should be 0 if ended empty).")
```

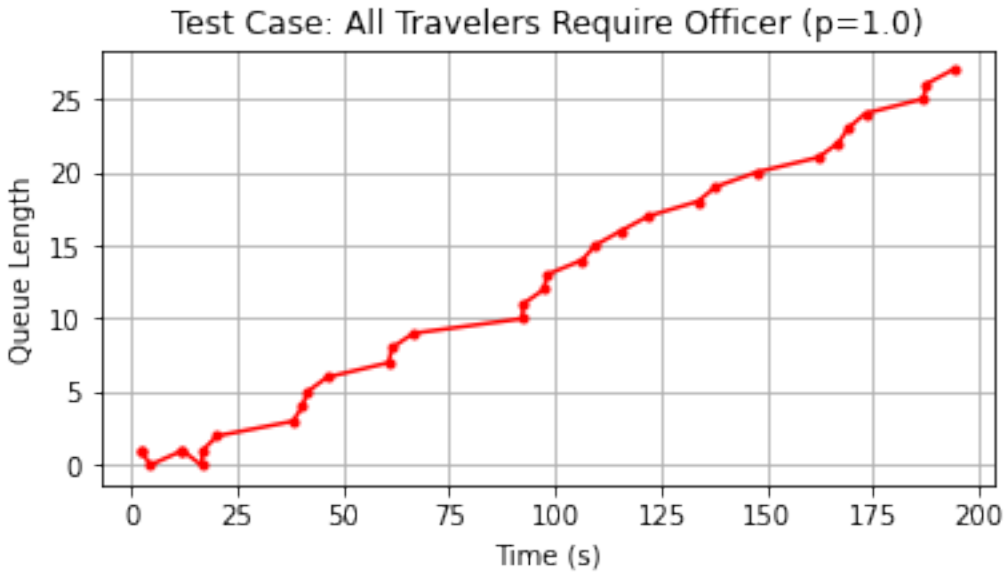
```
Final queue length after single arrival test: 0
Number of arrivals served: 0 (should be 0 if ended empty).
```

1.1.4 4.4 Test Case: Extra Screening Always Needed (Officer Logic Check)

Cell 8

[32]:
```
arrival_dist_medium = sts.expon(scale=6.0)  # ~0.1667 travelers/s
basic_service_dist_test = sts.expon(scale=5.0)  # average 5s basic screening
additional_service_dist_test = sts.truncnorm(a_add, b_add, loc=120.0, scale=120.
 ↪0)

airport_all_extra, schedule_all_extra = run_simulation_with_officer(
    arrival_dist_medium,
    basic_service_dist_test,
    additional_service_dist_test,
    run_until=200.0,
    prob_extra=1.0  # everyone needs the officer
)

plt.figure(figsize=(6,3))
plt.plot(airport_all_extra.queue.time_history, airport_all_extra.queue.
 ↪queue_length_history, 'r.-')
plt.xlabel("Time (s)")
plt.ylabel("Queue Length")
plt.title("Test Case: All Travelers Require Officer (p=1.0)")
plt.grid(True)
plt.show()
```

```
final_len_all_extra = airport_all_extra.queue.people_in_queue
print(f"Final queue length with officer p=1.0: {final_len_all_extra}")
```




```
Final queue length with officer p=1.0: 27
```

2 5. Empirical Analysis and Experiments

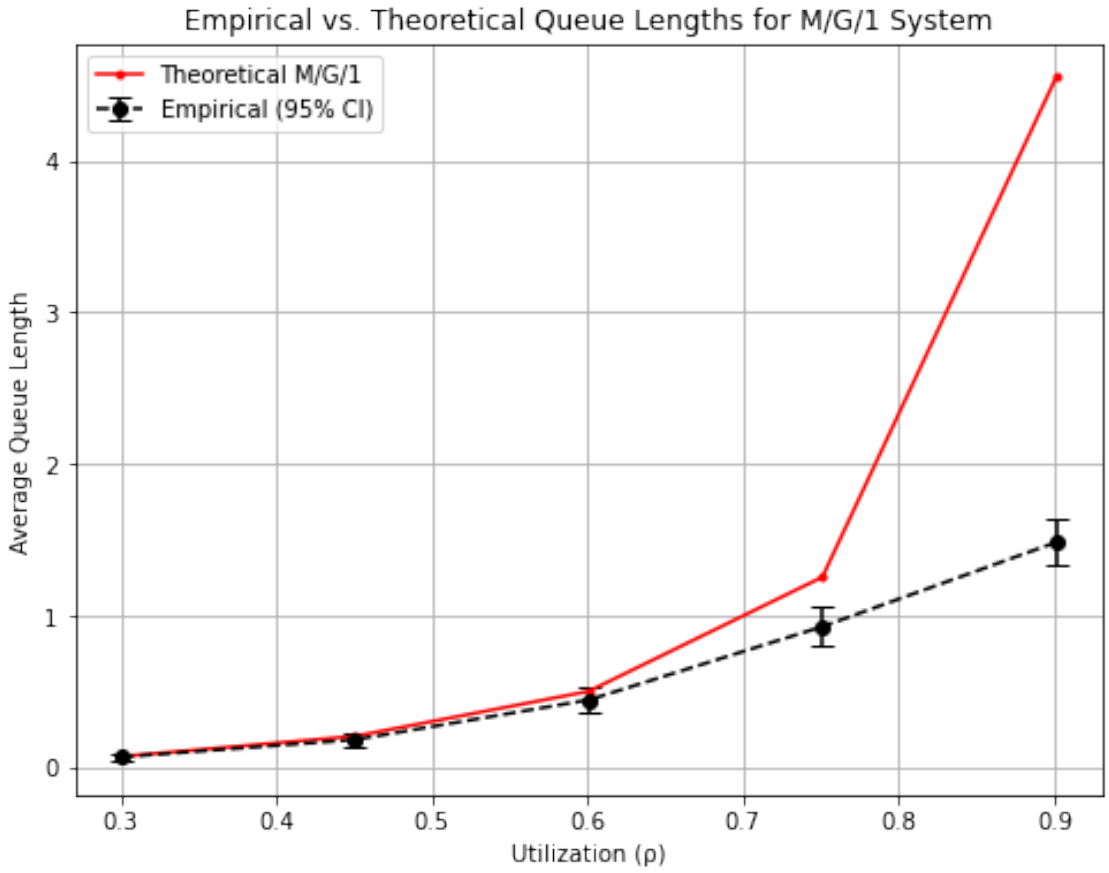
Cell 9

[253]:
```
##############################################################################
# 1) Helper function to estimate mean and variance of truncated normal(0,∞)
##############################################################################
def estimate_truncnorm_stats(mu, sigma, size=100_000):
    """
    Approximate E[T] and Var[T] for a truncated normal distribution
    truncated to (0,∞), with underlying normal loc=mu, scale=sigma.
    """
    a, b = (0 - mu)/sigma, np.inf
    dist = sts.truncnorm(a, b, loc=mu, scale=sigma)
    samples = dist.rvs(size=size)
    return np.mean(samples), np.var(samples, ddof=1)

##############################################################################
# 2) Function to run queue simulations across different arrival rates
##############################################################################
def run_experiment(arrival_rate_list, service_dist, run_until, num_trials=50):
    """
```

```
    Runs queue simulations for multiple arrival rates and returns:
      - Mean queue length
      - Standard error for queue length
    """
    results_mean, results_std_err = [], []

    for arrival_rate in tqdm(arrival_rate_list, desc="Running M/G/1␣
↪experiments"):
        arrival_distribution = sts.expon(scale=1.0 / arrival_rate)

        queue_lengths = []
        for _ in range(num_trials):
            airport, _ = run_simulation(arrival_distribution, service_dist,␣
↪run_until)
            queue_lengths.append(airport.queue.people_in_queue)

        results_mean.append(np.mean(queue_lengths))
        results_std_err.append(sts.sem(queue_lengths))

    return np.array(results_mean), np.array(results_std_err)

################################################################################
# 3) Theoretical M/G/1 formula for average queue length (Pollaczek-Khinchine)
################################################################################
def theoretical_mg1(rho, c_s_sq):
    """Returns expected queue length using M/G/1 formula."""
    return np.where(rho >= 1, np.inf, (rho**2 / (2.0 * (1.0 - rho))) * (1.0 +␣
↪c_s_sq))

################################################################################
# 4) Generate Error Plot (Empirical vs. Theoretical)
################################################################################
def make_error_plot(queue_type, rho, mean, std_err, theoretical_function,␣
↪c_s_sq):
    """Plots empirical queue lengths with confidence intervals vs. theoretical␣
↪values."""
    plt.figure(figsize=(8,6))
    plt.title(f'Empirical vs. Theoretical Queue Lengths for {queue_type}␣
↪System')
    plt.xlabel('Utilization ( )') #   =   * E[T_eff]
    plt.ylabel('Average Queue Length')

    # Empirical with 95% Confidence Interval
    ci = 1.96 * std_err
    plt.errorbar(rho, mean, ci, fmt='ko--', capsize=5, label='Empirical (95%␣
↪CI)')
```

```
    # Theoretical M/G/1
    plt.plot(rho, theoretical_function(rho, c_s_sq), 'r.-', label='Theoretical␣
 ↪M/G/1')

    plt.legend()
    plt.grid(True)
    plt.show()

################################################################################
# 5) Run the simulation and compare with theory
################################################################################
if __name__ == "__main__":
    arrival_rate_list = np.array([0.01, 0.015, 0.02, 0.025, 0.03])

    # Define service time distribution
    service_distribution = sts.truncnorm((0 - mu_service)/sigma_service, np.
 ↪inf, loc=mu_service, scale=sigma_service)

    # Run experiments
    run_until, num_trials = 300, 500
    mg1_mean, mg1_std_err = run_experiment(arrival_rate_list,␣
 ↪service_distribution, run_until, num_trials)

    # Compute theoretical parameters
    E_T, Var_T = estimate_truncnorm_stats(mu_service, sigma_service)
    c_s_sq = Var_T / (E_T**2)
    rho = arrival_rate_list * E_T

    # Generate final plot
    make_error_plot('M/G/1', rho, mg1_mean, mg1_std_err, theoretical_mg1,␣
 ↪c_s_sq)
```

```
Running M/G/1 experiments: 100%|          | 5/5 [00:01<00:00,  3.21it/s]
```



Cell 10

[237]:
```
# 1) Truncated normal stats was estimated before.

##############################################################################
# 2) Combine basic + additional screening into "effective" service-time stats
##############################################################################
def compute_effective_service_stats(
    p_extra,
    basic_mu, basic_sigma,
    add_mu, add_sigma
):
    """
    Approximate E[T_eff], Var[T_eff] for T_eff = T1 + Bernoulli(p_extra)*T2,
    using Monte Carlo for T1, T2.
    """
    E_T1, Var_T1 = estimate_truncnorm_stats(basic_mu, basic_sigma)
    E_T2, Var_T2 = estimate_truncnorm_stats(add_mu, add_sigma)

    # Mean
```

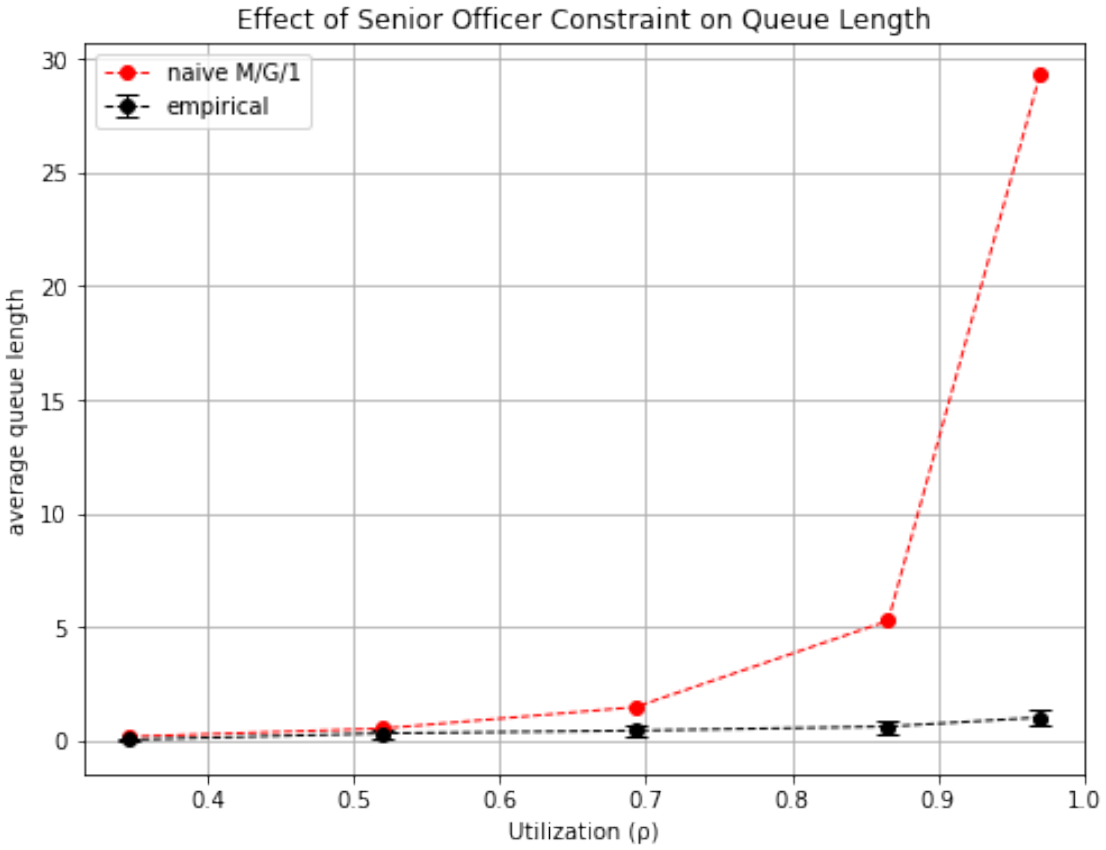
```
    E_Teff = E_T1 + p_extra * E_T2

    # Var
    # Var(T1 + X*T2) = Var(T1) + p_extra * Var(T2) +␣
 ↪p_extra(1-p_extra)*(E[T2])^2
    Var_Teff = Var_T1 + p_extra*Var_T2 + p_extra*(1-p_extra)*(E_T2**2)

    return E_Teff, Var_Teff

##############################################################################
# 3) Pollaczek-Khinchine (M/G/1) for average queue length
##############################################################################
def mg1_queue_length(rho, c_s_sq):
    """
    Lq = (rho^2 / [2(1-rho)]) * (1 + c_s_sq), vectorized over 'rho'.
    """
    out = []
    for r in rho:
        if r >= 1.0:
            out.append(np.inf)
        else:
            val = (r**2 / (2.0*(1.0 - r))) * (1.0 + c_s_sq)
            out.append(val)
    return np.array(out)

##############################################################################
# 4) run_experiment_officer: gather empirical results for single-officer␣
 ↪scenario
##############################################################################
def run_experiment_officer(
    arrival_rate_list,
    basic_mu, basic_sigma,       # for T1
    add_mu, add_sigma,           # for T2
    p_extra,
    run_until,
    num_trials=50
):
    """
    For each lambda in arrival_rate_list:
      - create arrival_dist = Exp(1/lambda)
      - basic_dist = truncated normal(basic_mu, basic_sigma)
      - additional_dist = truncated normal(add_mu, add_sigma)
      - run single-officer simulation multiple times
      - record final queue length
    Return arrays (mean_queue_length, std_err_queue_length).
    """
    results_mean = []
```

```
    results_std_err = []

    for lam in tqdm(arrival_rate_list, desc="Officer experiments"):
        arrival_distribution = sts.expon(scale=1.0 / lam)

        # basic screening
        a1, b1 = (0 - basic_mu)/basic_sigma, np.inf
        basic_service_dist = sts.truncnorm(a1, b1, loc=basic_mu,␣
 ↪scale=basic_sigma)

        # additional screening
        a2, b2 = (0 - add_mu)/add_sigma, np.inf
        add_service_dist = sts.truncnorm(a2, b2, loc=add_mu, scale=add_sigma)

        queue_lengths = []
        for _ in range(num_trials):
            airport, _ = run_simulation_with_officer(
                arrival_distribution, basic_service_dist, add_service_dist,
                run_until=run_until, prob_extra=p_extra
            )
            queue_lengths.append(airport.queue.people_in_queue)

        results_mean.append(np.mean(queue_lengths))
        results_std_err.append(sts.sem(queue_lengths))

    return np.array(results_mean), np.array(results_std_err)

##############################################################################
# 5) Plot function: empirical (officer scenario) vs. naive M/G/1 "effective"␣
 ↪theory
##############################################################################
def make_officer_plot(
    arrival_rate_list,
    mean_emp,
    std_emp,
    basic_mu, basic_sigma,
    add_mu, add_sigma,
    p_extra
):
    """
    1) Compute E[T_eff], Var[T_eff].
    2) For each arrival_rate, compute rho = lambda * E[T_eff].
    3) Plot empirical points vs. M/G/1 formula with that effective distribution.
    """
    # A) Effective stats
    E_Teff, Var_Teff = compute_effective_service_stats(
        p_extra,
```

```
        basic_mu, basic_sigma,
        add_mu, add_sigma
    )
    c_s_sq = Var_Teff / (E_Teff**2)

    # B) Compute rho array
    rhos = arrival_rate_list * E_Teff

    # C) Theoretical M/G/1
    theory_vals = mg1_queue_length(rhos, c_s_sq)

    # D) Plot
    plt.figure(figsize=(8,6))
    plt.title("Effect of Senior Officer Constraint on Queue Length")
    plt.xlabel("Utilization ( )")
    plt.ylabel("average queue length")

    ci = 1.96 * std_emp
    plt.errorbar(
        rhos, mean_emp, yerr=ci,
        color='black', marker='o', linestyle='--', linewidth=1,
        capsize=5, label='empirical'
    )
    plt.plot(
        rhos, theory_vals,
        color='red', marker='o', linestyle='--', linewidth=1,
        label='naive M/G/1'
    )

    plt.grid(True)
    plt.legend()
    plt.show()


##############################################################################
# 6) Example usage
##############################################################################
if __name__ == "__main__":
    # Let's pick arrival rates small enough that the naive   < 1
    # if E[T_eff] ~ 30 + 0.03*120= 33.6 => 1/E[T_eff] ~ 0.0297
    arrival_rates = np.array([0.01, 0.015, 0.02, 0.025, 0.028])

    # Basic screening: truncated normal( mu=30s, sigma=10s )
    basic_mu = 30.0
    basic_sigma = 10.0
```

```
    # Additional screening: truncated normal( mu=120s, sigma=120s ) => 2 min ±␣
↪2 min
    add_mu = 120.0
    add_sigma = 120.0

    # Probability that a traveler needs additional check
    p_extra = 0.03

    run_until = 300
    num_trials = 50

    # 1) Gather empirical data
    officer_mean, officer_std_err = run_experiment_officer(
        arrival_rates,
        basic_mu, basic_sigma,
        add_mu, add_sigma,
        p_extra,
        run_until,
        num_trials
    )
    print("Officer experiment done.\n")

    # 2) Plot
    make_officer_plot(
        arrival_rates,
        officer_mean,
        officer_std_err,
        basic_mu, basic_sigma,
        add_mu, add_sigma,
        p_extra
    )
```

```
Officer experiments: 100%|          | 5/5 [00:00<00:00, 24.99it/s]

Officer experiment done.
```



3 6. Steady-State Analysis and Transient State Effects

Cell 11

[255]:
```
################################################################################
# 1) Helper: Time-average queue length post-warmup
################################################################################
def time_average_queue_length(time_array, length_array, warmup=0.0):
    data = list(zip(time_array, length_array))
    data.sort(key=lambda x: x[0])  # ensure sorted in time
    data = [d for d in data if d[0] >= warmup]
    if len(data) < 2:
        return 0.0

    total_area = 0.0
    total_time = 0.0
    for i in range(len(data) - 1):
        t1, L1 = data[i]
        t2, L2 = data[i+1]
        dt = t2 - t1
```

```
        avg_height = 0.5 * (L1 + L2)
        total_area += avg_height * dt
        total_time += dt

    return total_area / total_time if total_time > 0 else 0.0

################################################################################
# 2) Main function: multiple runs, time-average the queue length after warm-up
################################################################################
def gather_steady_state_data(
    param_list,
    run_simulation_func,  # e.g. run_simulation(...) returning (airport,␣
 ↪schedule)
    build_distributions_func, # returns (arrival_dist, service_dist)
    warmup=3600.0,        # discard the first hour
    run_until=28800.0,    # run for 8 hours
    num_reps=5,
    seed=123
):
    np.random.seed(seed)
    means = []
    errs = []

    for param in tqdm(param_list, desc="Steady-State experiments"):
        avg_list = []
        for _ in range(num_reps):
            arrival_dist, service_dist = build_distributions_func(param)
            airport, _ = run_simulation_func(arrival_dist, service_dist,␣
 ↪run_until)

            times = np.array(airport.queue.time_history)
            lengths = np.array(airport.queue.queue_length_history)
            avg_q = time_average_queue_length(times, lengths, warmup=warmup)
            avg_list.append(avg_q)

        mean_q = np.mean(avg_list)
        sem_q = sts.sem(avg_list)

        means.append(mean_q)
        errs.append(sem_q)

    return np.array(means), np.array(errs)

################################################################################
# 3) Pollaczek-Khinchine formula for M/G/1
################################################################################
def mg1_theoretical(rho, c_s_sq):
```

```
    out = []
    for r in rho:
        if r >= 1.0:
            out.append(np.inf)
        else:
            val = (r**2 / (2*(1-r))) * (1 + c_s_sq)
            out.append(val)
    return np.array(out)


if __name__ == "__main__":
    # A) Standard M/G/1 scenario with truncated normal service times
    import scipy.stats as sts

    # A1) Build function for arrival_dist, service_dist
    def build_example_distributions(lam):
        arrival_dist = sts.expon(scale=1.0 / lam)

        # truncated normal for basic service
        mu, sigma = 30.0, 10.0
        a, b = (0 - mu)/sigma, np.inf
        service_dist = sts.truncnorm(a, b, loc=mu, scale=sigma)
        return arrival_dist, service_dist

    #     run_simulation(arrival_dist, service_dist, run_until).
    def run_simulation_example(arrival_dist, service_dist, run_until):
        airport, schedule = run_simulation(arrival_dist, service_dist,␣
 ↪run_until)
        return airport, schedule

    # B) Pick arrival rates under 1/E[T] => ~0.033 if E[T]~30
    arrival_rates = [0.01, 0.015, 0.02, 0.025, 0.03]

    # We'll run for 22hrs, discard first two hour=3600s
    warmup = 2*3600
    run_until = 22*3600
    num_reps = 10

    # C) Gather empirical results
    means, errs = gather_steady_state_data(
        param_list=arrival_rates,
        run_simulation_func=run_simulation_example,
        build_distributions_func=build_example_distributions,
        warmup=warmup,
        run_until=run_until,
        num_reps=num_reps,
        seed=42
```

```
    )

    E_T, Var_T = estimate_truncnorm_stats(30.0, 10.0)
    c_s_sq = Var_T / (E_T**2)
    rhos = np.array(arrival_rates) * E_T
    theory_vals = mg1_theoretical(rhos, c_s_sq)

    ci = 1.96 * errs
    plt.figure(figsize=(8,5))
    plt.title("Steady-State Behavior Without the Senior Officer")
    plt.xlabel("Utilization") #(approx) =   * E[T]
    plt.ylabel("Time-Averaged Q length")

    # Empirical
    plt.errorbar(rhos, means, yerr=ci,
                 fmt='ko--', capsize=5, label='Time-Avg Q after warmup')
    # Theoretical
    plt.plot(rhos, theory_vals, 'r.-', label='M/G/1 Theory')

    plt.grid(True)
    plt.legend()
    plt.show()
```

```
Steady-State experiments: 100%|          | 5/5 [00:13<00:00,  2.69s/it]
```

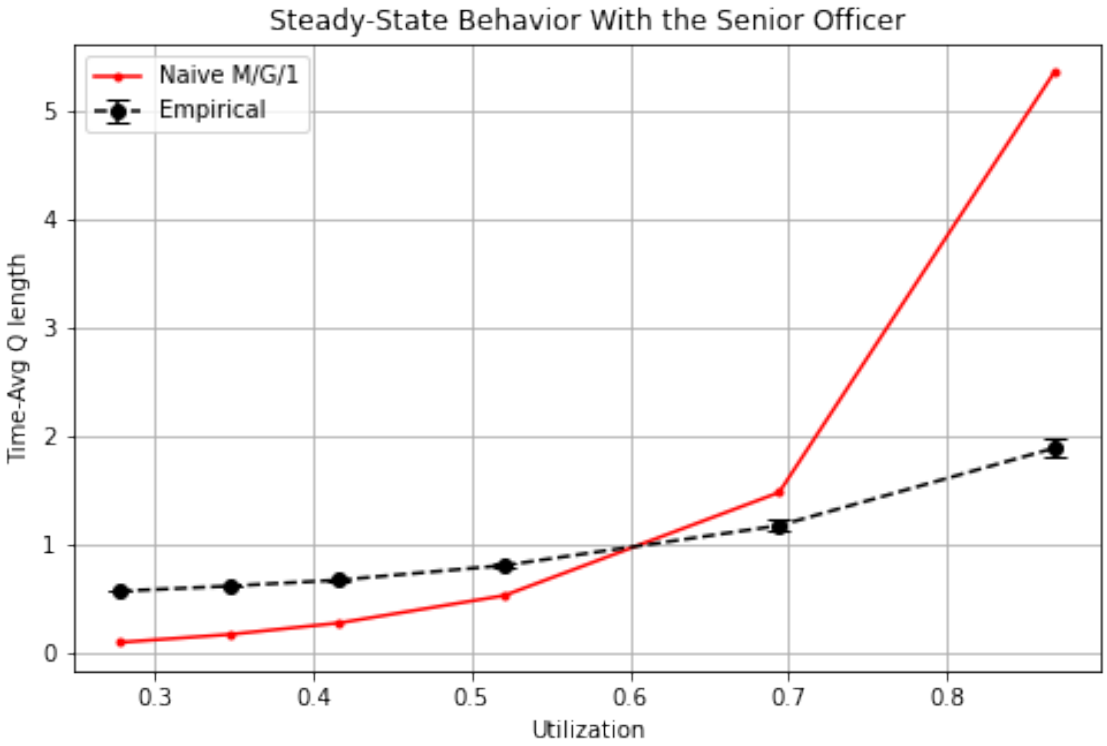
Cell 12

[257]:
```
##############################################################################
# 1) Time-Averaging & M/G/1 Theory
##############################################################################
def time_average_queue_length(time_array, length_array, warmup=0.0):
    data = sorted(zip(time_array, length_array), key=lambda x: x[0])
    data = [d for d in data if d[0] >= warmup]
    if len(data) < 2:
        return 0.0
    total_area, total_time = 0.0, 0.0
    for i in range(len(data) - 1):
        t1, L1 = data[i]
        t2, L2 = data[i+1]
        dt = t2 - t1
        avg_ht = 0.5*(L1 + L2)
        total_area += avg_ht*dt
        total_time += dt
    return total_area/total_time if total_time>0 else 0.0

def mg1_theoretical(rho, c_s_sq):
    out = []
    for r in rho:
        if r >= 1.0:
            out.append(np.inf)
        else:
            val = (r**2 / (2*(1-r))) * (1 + c_s_sq)
            out.append(val)
    return np.array(out)

def compute_effective_service_stats(p_extra, basic_mu, basic_sigma, add_mu,␣
 ↪add_sigma):
    E_T1, Var_T1 = estimate_truncnorm_stats(basic_mu, basic_sigma)
    E_T2, Var_T2 = estimate_truncnorm_stats(add_mu, add_sigma)
    E_Teff = E_T1 + p_extra*E_T2
    Var_Teff = Var_T1 + p_extra*Var_T2 + p_extra*(1-p_extra)*(E_T2**2)
    return E_Teff, Var_Teff

##############################################################################
# 2) Gathering Officer Data Over Long Runs
##############################################################################
def gather_officer_long_run(
    arrival_rates,
    run_simulation_officer_func,  # run_simulation_with_officer
    basic_mu, basic_sigma,
    add_mu, add_sigma,
```

```
    p_extra=0.03,
    run_until=28800,      # e.g., 8 hours
    warmup=3600,          # discard first hour
    num_reps=10,
    seed=42
):
    np.random.seed(seed)
    means = []
    errs = []

    for lam in tqdm(arrival_rates, desc="Long-run officer scenario"):
        avg_list = []
        for _ in range(num_reps):
            # Build arrival & service distributions
            arrival_dist = sts.expon(scale=1.0/lam)

            a_b, b_b = (0 - basic_mu)/basic_sigma, np.inf
            basic_sd = sts.truncnorm(a_b, b_b, loc=basic_mu, scale=basic_sigma)

            a_a, b_a = (0 - add_mu)/add_sigma, np.inf
            add_sd = sts.truncnorm(a_a, b_a, loc=add_mu, scale=add_sigma)

            # Run a single replication
            airport, _ = run_simulation_officer_func(
                arrival_dist, basic_sd, add_sd,
                run_until=run_until,
                prob_extra=p_extra
            )

            # Time-average queue length post-warmup
            times = np.array(airport.queue.time_history)
            lengths = np.array(airport.queue.queue_length_history)
            avg_q = time_average_queue_length(times, lengths, warmup)
            avg_list.append(avg_q)

        # Aggregate over reps
        m = np.mean(avg_list)
        s = sts.sem(avg_list)
        means.append(m)
        errs.append(s)

    return np.array(means), np.array(errs)

##############################################################################
# 3) Example Usage (Single Officer)
##############################################################################
if __name__ == "__main__":
```

```
    # Suppose arrival rates near the stable region
    arrival_rates = [0.008, 0.01, 0.012, 0.015, 0.02, 0.025]

    # Basic screening ~ truncated normal(30,10)
    basic_mu, basic_sigma = 30.0, 10.0
    # Additional screening ~ truncated normal(120,120)
    add_mu, add_sigma = 120.0, 120.0
    p_extra = 0.03

    # 1) Gather time-averaged queue length for 8-hour runs, discarding 1 hour␣
↪warmup
    means, errs = gather_officer_long_run(
        arrival_rates,
        run_simulation_with_officer,
        basic_mu, basic_sigma,
        add_mu, add_sigma,
        p_extra=p_extra,
        run_until=3600*22,  # 22 hours
        warmup=3606*2,      # discard first two hour
        num_reps=10,
        seed=42
    )

    # 2) Compute naive M/G/1 is already computed before
    E_Teff, Var_Teff = compute_effective_service_stats(
        p_extra,
        basic_mu, basic_sigma,
        add_mu, add_sigma
    )
    c_s_sq = Var_Teff / (E_Teff**2)

    rhos = np.array(arrival_rates) * E_Teff
    theory_vals = mg1_theoretical(rhos, c_s_sq)

    # 3) Plot
    ci = 1.96*errs
    plt.figure(figsize=(8,5))
    plt.title("Steady-State Behavior With the Senior Officer")
    plt.xlabel("Utilization") # ( * E[T_eff])
    plt.ylabel("Time-Avg Q length")

    plt.errorbar(rhos, means, yerr=ci, fmt='ko--', capsize=5, label="Empirical")
    plt.plot(rhos, theory_vals, 'r.-', label="Naive M/G/1")
    plt.grid(True)
    plt.legend()
    plt.show()
```

```
Long-run officer scenario: 100%|          | 6/6 [00:11<00:00,  1.89s/it]
```



4 Debugging attempt

Cell 13

[192]:
```python
def run_experiment_day_fixed(
    num_service_stations_list,
    run_simulation_func,
    build_distributions_func,
    run_until=3600*22,  # 22 hours
    num_reps=5,  # Reduce for debugging
    seed=123
):
    """
    Runs simulations for different numbers of service stations.
    Fixes:
      - Ensures travelers are being properly removed from queue
      - Verifies queue length remains reasonable
      - Prints queue health at regular intervals
    """
    np.random.seed(seed)
    results = {
```

```
        "num_stations": [],
        "mean_wait_time": [],
        "std_err_wait_time": [],
        "mean_queue_length": [],
        "std_err_queue_length": [],
        "max_queue_length": []
    }

    for num_stations in tqdm(num_service_stations_list, desc="Running␣
↪simulations"):
        wait_times, queue_lengths, max_lengths = [], [], []

        for rep in range(num_reps):
            print(f"\n Debugging Run {rep+1}/{num_reps} for {num_stations}␣
↪stations")

            arrival_dist, service_dist = build_distributions_func(num_stations)
            airport, _ = run_simulation_func(arrival_dist, service_dist,␣
↪run_until)

            travelers = airport.queue.traveler_records

            # DEBUG: Print a few traveler records to verify they are being␣
↪processed
            print(" Traveler Records Sample:")
            for rec in travelers[:5]:
                print(f"Arrival: {rec['arrival_t']:.2f}, Start: {rec['start_t']:
↪.2f}, Done: {rec['done_t']:.2f}")

            # Check queue length every 1000 seconds to see if it remains␣
↪controlled
            if len(airport.queue.queue_length_history) > 0:
                sampled_times = np.linspace(0, run_until, num=5).astype(int)
                sampled_lengths = [
                    airport.queue.queue_length_history[min(i, len(airport.queue.
↪queue_length_history)-1)]
                    for i in sampled_times
                ]
                print(f" Queue Length at Intervals: {sampled_lengths}")

            # Calculate waiting time correctly
            computed_wait_times = [(rec['start_t'] - rec['arrival_t']) for rec␣
↪in travelers if rec['start_t'] is not None]
            if len(computed_wait_times) == 0:
                print(" ERROR: No travelers recorded in queue!")
                continue  # Skip this run
```

```
            avg_wait_time = np.mean(computed_wait_times)
            avg_queue_length = np.mean(airport.queue.queue_length_history)
            max_queue_length = np.max(airport.queue.queue_length_history)

            wait_times.append(avg_wait_time)
            queue_lengths.append(avg_queue_length)
            max_lengths.append(max_queue_length)

        results["num_stations"].append(num_stations)
        results["mean_wait_time"].append(np.mean(wait_times))
        results["std_err_wait_time"].append(sts.sem(wait_times))
        results["mean_queue_length"].append(np.mean(queue_lengths))
        results["std_err_queue_length"].append(sts.sem(queue_lengths))
        results["max_queue_length"].append(np.mean(max_lengths))

        print(f" Num Stations: {num_stations}, Avg Wait Time:␣
 ↪{results['mean_wait_time'][-1]:.2f}, Queue Length:␣
 ↪{results['mean_queue_length'][-1]:.2f}")

    return results

def mg1_theoretical_metrics_fixed(num_service_stations_list, E_T, Var_T,␣
 ↪arrival_rate=10/60):
    """
    Computes M/G/1 theoretical values for waiting time and queue length.
    """
    c_s_sq = Var_T / (E_T**2)
    theoretical_results = {"num_stations": [], "Wq": [], "Lq": []}

    for num_stations in num_service_stations_list:
        rho = (arrival_rate / num_stations) * E_T
        if rho >= 1.0:
            theoretical_results["Wq"].append(np.inf)
            theoretical_results["Lq"].append(np.inf)
        else:
            Wq = (rho**2 / (2 * (1 - rho))) * (1 + c_s_sq) * E_T
            Lq = Wq * arrival_rate / num_stations
            theoretical_results["Wq"].append(Wq)
            theoretical_results["Lq"].append(Lq)
        theoretical_results["num_stations"].append(num_stations)

    return theoretical_results

def plot_comparison(empirical, theoretical, ylabel, title, key_empirical,␣
 ↪key_theoretical):
    """
```

```
    Generates a plot comparing empirical and theoretical values.
    """
    plt.figure(figsize=(8,5))
    plt.title(title)
    plt.xlabel("Number of Service Stations")
    plt.ylabel(ylabel)

    # Empirical results with error bars
    plt.errorbar(
        empirical["num_stations"], empirical[key_empirical],
        yerr=1.96 * np.array(empirical[key_empirical.replace("mean",␣
↪"std_err")]),
        fmt='ko--', capsize=5, label='Empirical (95% CI)'
    )

    # Theoretical results
    plt.plot(
        theoretical["num_stations"], theoretical[key_theoretical],
        'r.-', label='Theoretical M/G/1'
    )

    plt.grid(True)
    plt.legend()
    plt.show()

if __name__ == "__main__":
    num_service_stations_list = list(range(1, 11))  # Test from 1 to 10 stations

    # Estimate effective service time stats
    E_T, Var_T = estimate_truncnorm_stats(30.0, 10.0)

    # Run empirical simulations (no officer)
    empirical_results = run_experiment_day_fixed(
        num_service_stations_list,
        run_simulation,
        build_example_distributions,
        run_until=28800,  # 8 hours
        num_reps=3  # Reduce for debugging
    )

    theoretical_results =␣
↪mg1_theoretical_metrics_fixed(num_service_stations_list, E_T, Var_T)

    plot_comparison(empirical_results, theoretical_results, "Average Waiting␣
↪Time (s)", "Waiting Time vs. Service Stations", "mean_wait_time", "Wq")
    plot_comparison(empirical_results, theoretical_results, "Average Queue␣
↪Length", "Queue Length vs. Service Stations", "mean_queue_length", "Lq")
```

```
    plt.figure(figsize=(8,5))
    plt.title("Maximum Queue Length vs. Service Stations")
    plt.xlabel("Number of Service Stations")
    plt.ylabel("Max Queue Length")
    plt.plot(empirical_results["num_stations"],␣
 ↪empirical_results["max_queue_length"], 'bo-', label='Empirical')
    plt.grid(True)
    plt.legend()
    plt.show()
```

```
Running simulations:   0%|                                          | 0/10 [00:00<?, ?it/s]


  Debugging Run 1/3 for 1 stations
  Traveler Records Sample:
Arrival: 1.19, Start: 1.19, Done: 23.73
Arrival: 1.53, Start: 23.73, Done: 46.32
Arrival: 2.33, Start: 46.32, Done: 78.95
Arrival: 3.60, Start: 78.95, Done: 134.97
Arrival: 4.15, Start: 134.97, Done: 148.51
  Queue Length at Intervals: [1, 6531, 13047, 19579, 26109]

  Debugging Run 2/3 for 1 stations
  Traveler Records Sample:
Arrival: 2.02, Start: 2.02, Done: 36.98
Arrival: 2.25, Start: 36.98, Done: 79.47
Arrival: 2.46, Start: 79.47, Done: 108.11
Arrival: 2.65, Start: 108.11, Done: 145.73
Arrival: 3.73, Start: 145.73, Done: 173.32
  Queue Length at Intervals: [1, 6513, 13038, 19552, 26055]

  Debugging Run 3/3 for 1 stations
Running simulations:  10%|                                | 1/10 [00:02<00:23,  2.63s/it]
  Traveler Records Sample:
Arrival: 4.06, Start: 4.06, Done: 42.19
Arrival: 4.20, Start: 42.19, Done: 54.90
Arrival: 6.98, Start: 54.90, Done: 86.69
Arrival: 9.17, Start: 86.69, Done: 127.07
Arrival: 11.69, Start: 127.07, Done: 152.42
  Queue Length at Intervals: [1, 6504, 13020, 19563, 26097]
  Num Stations: 1, Avg Wait Time: 13899.60, Queue Length: 13889.16

  Debugging Run 1/3 for 2 stations
  Traveler Records Sample:
Arrival: 0.06, Start: 0.06, Done: 37.47
Arrival: 0.49, Start: 37.47, Done: 72.49
```

```
Arrival: 0.55, Start: 72.49, Done: 118.38
Arrival: 1.48, Start: 118.38, Done: 159.14
Arrival: 4.14, Start: 159.14, Done: 179.25
  Queue Length at Intervals: [1, 6876, 13719, 20565, 27408]

  Debugging Run 2/3 for 2 stations
  Traveler Records Sample:
Arrival: 0.46, Start: 0.46, Done: 47.51
Arrival: 1.00, Start: 47.51, Done: 73.82
Arrival: 1.47, Start: 73.82, Done: 93.33
Arrival: 1.49, Start: 93.33, Done: 137.52
Arrival: 3.46, Start: 137.52, Done: 165.36
  Queue Length at Intervals: [1, 6861, 13695, 20538, 27366]

  Debugging Run 3/3 for 2 stations

Running simulations:  20%|                      | 2/10 [00:07<00:30,  3.77s/it]

  Traveler Records Sample:
Arrival: 0.28, Start: 0.28, Done: 16.43
Arrival: 1.16, Start: 16.43, Done: 52.84
Arrival: 4.57, Start: 52.84, Done: 73.13
Arrival: 4.58, Start: 73.13, Done: 112.89
Arrival: 4.64, Start: 112.89, Done: 148.16
  Queue Length at Intervals: [1, 6849, 13704, 20544, 27393]
  Num Stations: 2, Avg Wait Time: 14128.25, Queue Length: 28295.72

  Debugging Run 1/3 for 3 stations
  Traveler Records Sample:
Arrival: 0.18, Start: 0.18, Done: 31.15
Arrival: 0.40, Start: 31.15, Done: 63.88
Arrival: 0.63, Start: 63.88, Done: 95.07
Arrival: 1.20, Start: 95.07, Done: 118.54
Arrival: 1.88, Start: 118.54, Done: 149.42
  Queue Length at Intervals: [1, 6966, 13932, 20892, 27864]

  Debugging Run 2/3 for 3 stations
  Traveler Records Sample:
Arrival: 1.70, Start: 1.70, Done: 11.97
Arrival: 2.26, Start: 11.97, Done: 46.88
Arrival: 2.51, Start: 46.88, Done: 69.59
Arrival: 2.67, Start: 69.59, Done: 94.10
Arrival: 2.72, Start: 94.10, Done: 139.64
  Queue Length at Intervals: [1, 6948, 13914, 20889, 27855]

  Debugging Run 3/3 for 3 stations

Running simulations:  30%|                      | 3/10 [00:13<00:35,  5.07s/it]

  Traveler Records Sample:
```

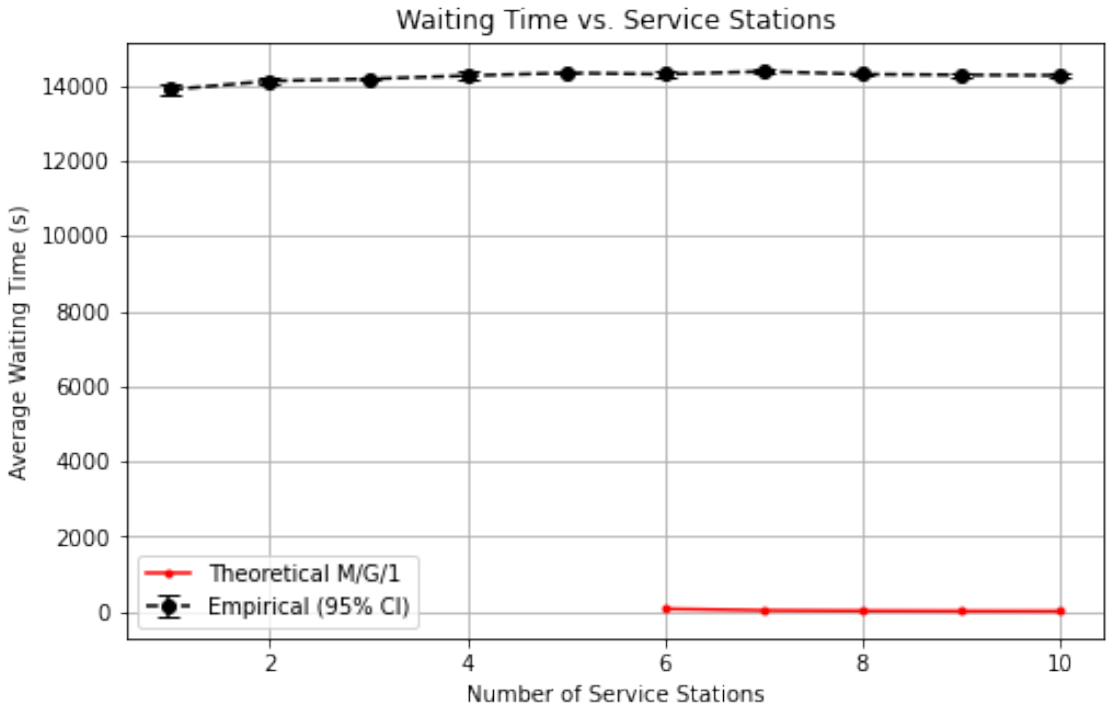
```
Arrival: 0.06, Start: 0.06, Done: 16.76
Arrival: 0.89, Start: 16.76, Done: 53.58
Arrival: 0.96, Start: 53.58, Done: 82.57
Arrival: 1.06, Start: 82.57, Done: 104.98
Arrival: 1.08, Start: 104.98, Done: 143.17
  Queue Length at Intervals: [1, 6966, 13941, 20895, 27852]
  Num Stations: 3, Avg Wait Time: 14182.23, Queue Length: 42749.63

  Debugging Run 1/3 for 4 stations
  Traveler Records Sample:
Arrival: 0.18, Start: 0.18, Done: 40.10
Arrival: 0.35, Start: 40.10, Done: 57.66
Arrival: 0.84, Start: 57.66, Done: 79.71
Arrival: 1.10, Start: 79.71, Done: 85.39
Arrival: 1.53, Start: 85.39, Done: 111.32
  Queue Length at Intervals: [1, 7020, 14049, 21069, 28083]

  Debugging Run 2/3 for 4 stations
  Traveler Records Sample:
Arrival: 0.37, Start: 0.37, Done: 31.02
Arrival: 0.45, Start: 31.02, Done: 60.31
Arrival: 1.67, Start: 60.31, Done: 92.15
Arrival: 1.81, Start: 92.15, Done: 114.40
Arrival: 1.89, Start: 114.40, Done: 144.69
  Queue Length at Intervals: [1, 7029, 14049, 21066, 28098]

  Debugging Run 3/3 for 4 stations
Running simulations:  40%|                   | 4/10 [00:22<00:39,  6.57s/it]
  Traveler Records Sample:
Arrival: 0.09, Start: 0.09, Done: 12.42
Arrival: 0.29, Start: 12.42, Done: 44.41
Arrival: 0.44, Start: 44.41, Done: 66.45
Arrival: 1.16, Start: 66.45, Done: 87.20
Arrival: 1.51, Start: 87.20, Done: 113.90
  Queue Length at Intervals: [1, 7014, 14034, 21063, 28095]
  Num Stations: 4, Avg Wait Time: 14269.35, Queue Length: 57209.23

  Debugging Run 1/3 for 5 stations
  Traveler Records Sample:
Arrival: 1.12, Start: 1.12, Done: 26.80
Arrival: 1.16, Start: 26.80, Done: 54.39
Arrival: 1.19, Start: 54.39, Done: 69.00
Arrival: 1.21, Start: 69.00, Done: 91.22
Arrival: 1.38, Start: 91.22, Done: 133.24
  Queue Length at Intervals: [1, 7062, 14127, 21180, 28239]

  Debugging Run 2/3 for 5 stations
```

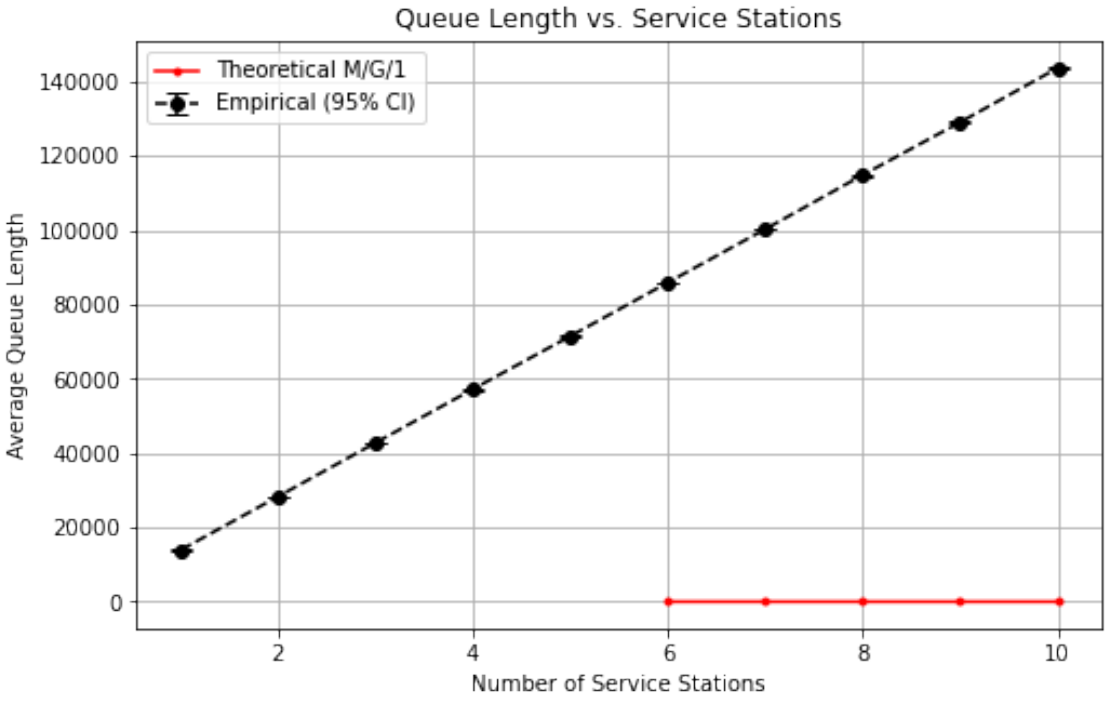
```
  Traveler Records Sample:
Arrival: 0.04, Start: 0.04, Done: 36.82
Arrival: 0.06, Start: 36.82, Done: 79.58
Arrival: 0.11, Start: 79.58, Done: 100.92
Arrival: 0.14, Start: 100.92, Done: 126.22
Arrival: 0.30, Start: 126.22, Done: 175.05
  Queue Length at Intervals: [1, 7068, 14115, 21171, 28233]

  Debugging Run 3/3 for 5 stations
Running simulations:  50%|                  | 5/10 [00:33<00:40,  8.10s/it]
  Traveler Records Sample:
Arrival: 0.11, Start: 0.11, Done: 46.20
Arrival: 0.12, Start: 46.20, Done: 82.31
Arrival: 0.19, Start: 82.31, Done: 103.06
Arrival: 0.24, Start: 103.06, Done: 139.05
Arrival: 0.58, Start: 139.05, Done: 173.05
  Queue Length at Intervals: [1, 7059, 14106, 21171, 28245]
  Num Stations: 5, Avg Wait Time: 14342.21, Queue Length: 71475.48

  Debugging Run 1/3 for 6 stations
  Traveler Records Sample:
Arrival: 0.02, Start: 0.02, Done: 12.05
Arrival: 0.37, Start: 12.05, Done: 34.45
Arrival: 0.54, Start: 34.45, Done: 56.48
Arrival: 0.78, Start: 56.48, Done: 101.75
Arrival: 1.09, Start: 101.75, Done: 116.80
  Queue Length at Intervals: [1, 7077, 14166, 21246, 28320]

  Debugging Run 2/3 for 6 stations
  Traveler Records Sample:
Arrival: 0.14, Start: 0.14, Done: 23.63
Arrival: 0.28, Start: 23.63, Done: 57.72
Arrival: 0.51, Start: 57.72, Done: 102.77
Arrival: 0.63, Start: 102.77, Done: 135.45
Arrival: 0.70, Start: 135.45, Done: 166.09
  Queue Length at Intervals: [1, 7080, 14160, 21255, 28332]

  Debugging Run 3/3 for 6 stations
Running simulations:  60%|                 | 6/10 [00:46<00:39,  9.77s/it]
  Traveler Records Sample:
Arrival: 0.06, Start: 0.06, Done: 23.12
Arrival: 0.09, Start: 23.12, Done: 51.25
Arrival: 0.14, Start: 51.25, Done: 77.33
Arrival: 0.24, Start: 77.33, Done: 91.17
Arrival: 0.46, Start: 91.17, Done: 136.16
  Queue Length at Intervals: [1, 7083, 14163, 21237, 28314]
```

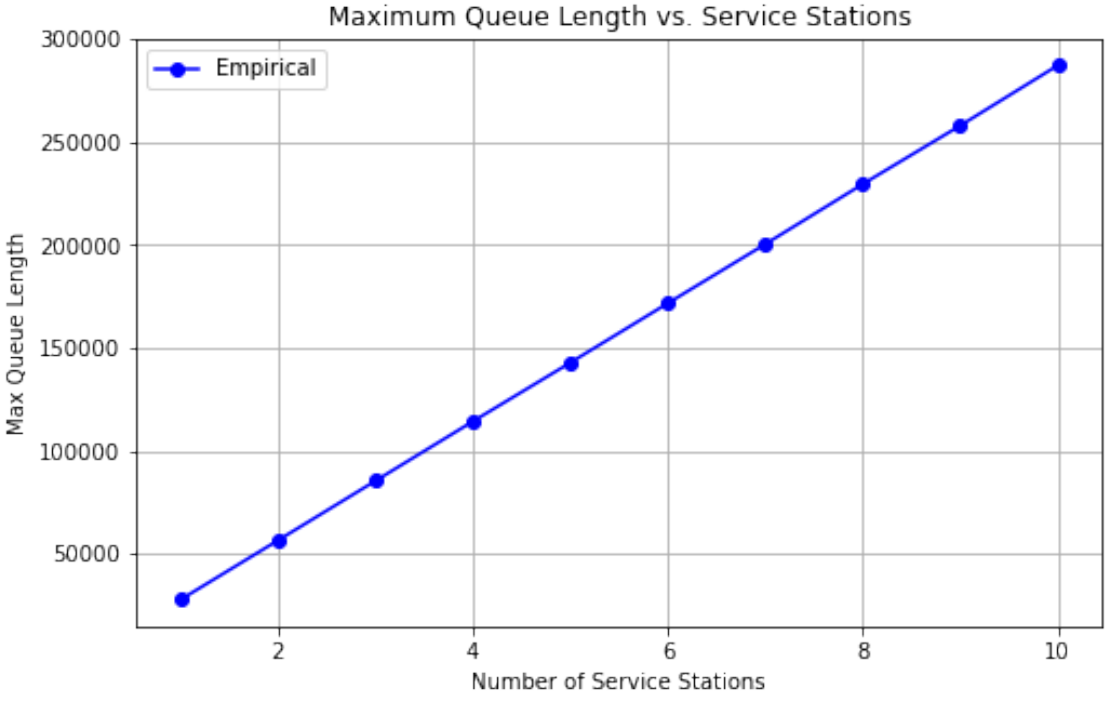
```
  Num Stations: 6, Avg Wait Time: 14307.98, Queue Length: 85904.55

  Debugging Run 1/3 for 7 stations
  Traveler Records Sample:
Arrival: 0.09, Start: 0.09, Done: 37.17
Arrival: 0.10, Start: 37.17, Done: 60.18
Arrival: 0.11, Start: 60.18, Done: 96.32
Arrival: 0.34, Start: 96.32, Done: 133.79
Arrival: 0.53, Start: 133.79, Done: 142.17
  Queue Length at Intervals: [1, 7101, 14199, 21297, 28398]

  Debugging Run 2/3 for 7 stations
  Traveler Records Sample:
Arrival: 0.01, Start: 0.01, Done: 11.63
Arrival: 0.07, Start: 11.63, Done: 36.84
Arrival: 0.33, Start: 36.84, Done: 67.33
Arrival: 0.54, Start: 67.33, Done: 113.68
Arrival: 0.64, Start: 113.68, Done: 121.78
  Queue Length at Intervals: [1, 7098, 14187, 21288, 28395]

  Debugging Run 3/3 for 7 stations
Running simulations:  70%|               | 7/10 [01:01<00:34, 11.58s/it]
  Traveler Records Sample:
Arrival: 0.10, Start: 0.10, Done: 11.53
Arrival: 0.18, Start: 11.53, Done: 58.34
Arrival: 0.25, Start: 58.34, Done: 82.40
Arrival: 0.31, Start: 82.40, Done: 106.46
Arrival: 0.53, Start: 106.46, Done: 141.16
  Queue Length at Intervals: [1, 7096, 14190, 21291, 28398]
  Num Stations: 7, Avg Wait Time: 14381.31, Queue Length: 100267.60

  Debugging Run 1/3 for 8 stations
  Traveler Records Sample:
Arrival: 0.17, Start: 0.17, Done: 23.36
Arrival: 0.38, Start: 23.36, Done: 50.88
Arrival: 0.43, Start: 50.88, Done: 85.37
Arrival: 0.55, Start: 85.37, Done: 122.71
Arrival: 0.58, Start: 122.71, Done: 155.46
  Queue Length at Intervals: [1, 7113, 14226, 21333, 28449]

  Debugging Run 2/3 for 8 stations
  Traveler Records Sample:
Arrival: 0.18, Start: 0.18, Done: 34.40
Arrival: 0.38, Start: 34.40, Done: 69.14
Arrival: 0.52, Start: 69.14, Done: 101.91
Arrival: 0.60, Start: 101.91, Done: 139.68
Arrival: 0.87, Start: 139.68, Done: 180.10
```

```
  Queue Length at Intervals: [1, 7113, 14217, 21333, 28440]

  Debugging Run 3/3 for 8 stations
Running simulations:  80%|              | 8/10 [01:19<00:26, 13.42s/it]
  Traveler Records Sample:
Arrival: 0.12, Start: 0.12, Done: 33.27
Arrival: 0.13, Start: 33.27, Done: 52.97
Arrival: 0.15, Start: 52.97, Done: 79.36
Arrival: 0.23, Start: 79.36, Done: 94.37
Arrival: 0.77, Start: 94.37, Done: 136.47
  Queue Length at Intervals: [1, 7110, 14214, 21318, 28437]
  Num Stations: 8, Avg Wait Time: 14308.60, Queue Length: 114844.69

  Debugging Run 1/3 for 9 stations
  Traveler Records Sample:
Arrival: 0.24, Start: 0.24, Done: 49.53
Arrival: 0.31, Start: 49.53, Done: 87.32
Arrival: 0.35, Start: 87.32, Done: 111.12
Arrival: 0.44, Start: 111.12, Done: 143.23
Arrival: 0.48, Start: 143.23, Done: 172.84
  Queue Length at Intervals: [1, 7116, 14235, 21351, 28467]

  Debugging Run 2/3 for 9 stations
  Traveler Records Sample:
Arrival: 0.07, Start: 0.07, Done: 15.81
Arrival: 0.27, Start: 15.81, Done: 55.84
Arrival: 0.27, Start: 55.84, Done: 108.32
Arrival: 0.33, Start: 108.32, Done: 140.59
Arrival: 0.53, Start: 140.59, Done: 184.50
  Queue Length at Intervals: [1, 7122, 14247, 21366, 28482]

  Debugging Run 3/3 for 9 stations
Running simulations:  90%|             | 9/10 [01:38<00:15, 15.30s/it]
  Traveler Records Sample:
Arrival: 0.00, Start: 0.00, Done: 33.82
Arrival: 0.02, Start: 33.82, Done: 80.53
Arrival: 0.02, Start: 80.53, Done: 105.42
Arrival: 0.03, Start: 105.42, Done: 146.99
Arrival: 0.15, Start: 146.99, Done: 168.65
  Queue Length at Intervals: [1, 7122, 14244, 21366, 28485]
  Num Stations: 9, Avg Wait Time: 14288.44, Queue Length: 129048.34

  Debugging Run 1/3 for 10 stations
  Traveler Records Sample:
Arrival: 0.56, Start: 0.56, Done: 36.88
Arrival: 0.69, Start: 36.88, Done: 83.99
```

```
Arrival: 0.77, Start: 83.99, Done: 120.14
Arrival: 0.83, Start: 120.14, Done: 145.80
Arrival: 0.88, Start: 145.80, Done: 174.44
  Queue Length at Intervals: [1, 7135, 14259, 21387, 28518]

  Debugging Run 2/3 for 10 stations
  Traveler Records Sample:
Arrival: 0.41, Start: 0.41, Done: 37.77
Arrival: 0.67, Start: 37.77, Done: 45.88
Arrival: 0.68, Start: 45.88, Done: 69.01
Arrival: 0.68, Start: 69.01, Done: 105.23
Arrival: 0.74, Start: 105.23, Done: 135.18
  Queue Length at Intervals: [1, 7119, 14250, 21384, 28506]

  Debugging Run 3/3 for 10 stations
Running simulations: 100%|           | 10/10 [02:00<00:00, 12.03s/it]
  Traveler Records Sample:
Arrival: 0.04, Start: 0.04, Done: 31.33
Arrival: 0.04, Start: 31.33, Done: 40.36
Arrival: 0.09, Start: 40.36, Done: 69.06
Arrival: 0.09, Start: 69.06, Done: 92.75
Arrival: 0.11, Start: 92.75, Done: 110.99
  Queue Length at Intervals: [1, 7122, 14259, 21385, 28518]
  Num Stations: 10, Avg Wait Time: 14277.90, Queue Length: 143660.96
```




Queue Length vs. Service Stations
Theoretical M/G/1
Empirical (95% CI)
Average Queue Length
0
20000
40000
60000
80000
100000
120000
140000
2
4
6
8
10
Number of Service Stations


Maximum Queue Length vs. Service Stations
Empirical
Max Queue Length
50000
100000
150000
200000
250000
300000
2
4
6
8
10
Number of Service Stations

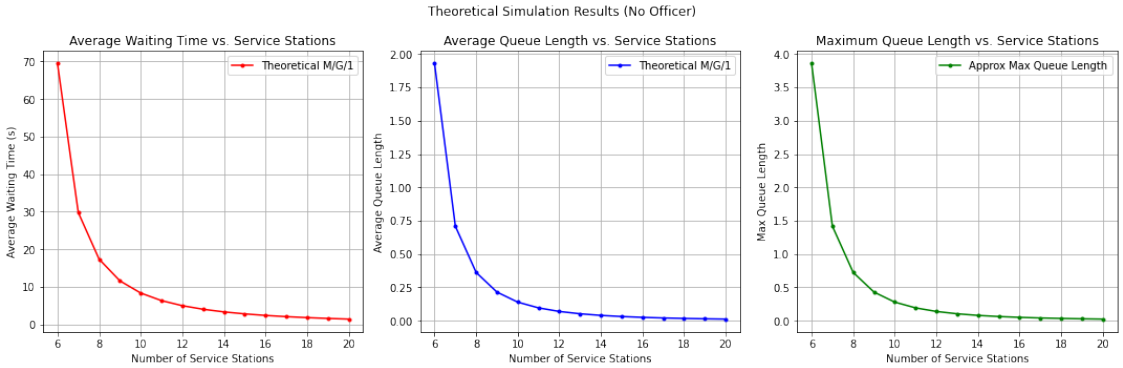
5 Service Station Optimization: Theoretical vs. Empirical Analysis

5.0.1 Theoretical - Without Officer

Cell 14

[251]:
```
### Parameters (will use for all the code cells after this)
arrival_rate = 10 / 60  # 10 travelers per minute → arrivals per second
E_T = 30.0  # Mean service time (seconds)
Var_T = 100.0  # Variance of service time (std dev squared)

# Compute coefficient of variation squared (c_s^2)
c_s_sq = Var_T / (E_T ** 2)

# Define range of service stations to analyze
num_service_stations_list = np.arange(1, 21)

# Compute theoretical metrics for each service station count
theoretical_results = {
    "num_stations": [],
    "Wq": [],
    "Lq": [],
    "MaxQueueLength": []  # Approximation for max queue length
}

for num_stations in num_service_stations_list:
    rho = (arrival_rate / num_stations) * E_T  # Utilization factor

    if rho >= 1.0:
        # System is unstable, waiting time and queue length go to infinity
        Wq = np.inf
        Lq = np.inf
        MaxQueueLength = np.inf
    else:
        Wq = (rho**2 / (2 * (1 - rho))) * (1 + c_s_sq) * E_T  # Waiting time
        Lq = Wq * (arrival_rate / num_stations)  # Queue length
        MaxQueueLength = Lq * 2  # Rough approximation for max queue length

    theoretical_results["num_stations"].append(num_stations)
    theoretical_results["Wq"].append(Wq)
    theoretical_results["Lq"].append(Lq)
    theoretical_results["MaxQueueLength"].append(MaxQueueLength)

# Plot Results
fig, axes = plt.subplots(1, 3, figsize=(15,5))

# 1) Average Waiting Time
axes[0].plot(
```

```
    theoretical_results["num_stations"], theoretical_results["Wq"],
    'r.-', label='Theoretical M/G/1'
)
axes[0].set_xlabel("Number of Service Stations")
axes[0].set_ylabel("Average Waiting Time (s)")
axes[0].set_title("Average Waiting Time vs. Service Stations")
axes[0].grid(True)
axes[0].legend()

# 2) Average Queue Length
axes[1].plot(
    theoretical_results["num_stations"], theoretical_results["Lq"],
    'b.-', label='Theoretical M/G/1'
)
axes[1].set_xlabel("Number of Service Stations")
axes[1].set_ylabel("Average Queue Length")
axes[1].set_title("Average Queue Length vs. Service Stations")
axes[1].grid(True)
axes[1].legend()

# 3) Maximum Queue Length (Approximate)
axes[2].plot(
    theoretical_results["num_stations"], theoretical_results["MaxQueueLength"],
    'g.-', label='Approx Max Queue Length'
)
axes[2].set_xlabel("Number of Service Stations")
axes[2].set_ylabel("Max Queue Length")
axes[2].set_title("Maximum Queue Length vs. Service Stations")
axes[2].grid(True)
axes[2].legend()

plt.suptitle("Theoretical Simulation Results (No Officer)")
plt.tight_layout()
plt.show()
```



5.0.2 Emperical - Without officer

Cell 15

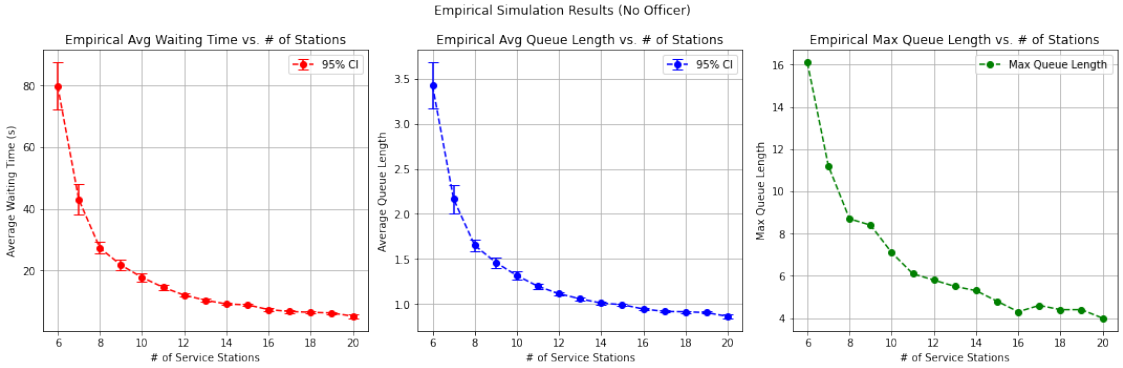
[254]:
```
def run_experiment_stations(num_stations_list, arrival_rate, service_dist,␣
↪warmup, run_time, num_trials=10):
    """
    Runs multiple replications of the airport security simulation for different
    numbers of service stations, with a warm-up period.
    Returns empirical means and 95% confidence intervals for:
        - Average waiting time Wq
        - Average queue length Lq
        - Maximum observed queue length
    """
    results = {
        "num_stations": [],
        "mean_wait_time": [],
        "std_err_wait_time": [],
        "mean_queue_length": [],
        "std_err_queue_length": [],
        "max_queue_length": [],
    }

    for num_stations in tqdm(num_stations_list, desc="Simulating different␣
↪service station counts"):
        all_Wq, all_Lq, all_MaxQ = [], [], []

        # Compute utilization  (should be < 1 for stability)
        E_T = service_dist.mean()  # Expected service time
        rho = (arrival_rate / num_stations) * E_T  # Utilization factor per␣
↪station
        if rho >= 1:
            print(f"Skipping {num_stations} stations ( ={rho:.2f}  1, unstable).
↪")
            continue  # Skip unstable cases

        for _ in range(num_trials):
            # Adjust arrival rate per station to avoid overload
            per_station_lambda = arrival_rate / num_stations
            arrival_dist = sts.expon(scale=1.0 / per_station_lambda)

            # Run the simulation for (warmup + steady-state time)
            total_time = warmup + run_time
            airport, _ = run_simulation(arrival_dist, service_dist, total_time)

            # Extract queue data
```

```
            time_array = np.array(airport.queue.time_history)
            length_array = np.array(airport.queue.queue_length_history)

            # Filter only steady-state data (after warm-up period)
            length_array = length_array[time_array >= warmup]
            travelers = [rec for rec in airport.queue.traveler_records if␣
 ↪rec['arrival_t'] >= warmup and rec['start_t'] is not None]

            # Compute Metrics
            avg_queue_length = np.mean(length_array) if len(length_array) > 0␣
 ↪else 0
            max_queue_length = np.max(length_array) if len(length_array) > 0␣
 ↪else 0
            avg_waiting_time = np.mean([rec['start_t'] - rec['arrival_t'] for␣
 ↪rec in travelers]) if travelers else 0

            all_Lq.append(avg_queue_length)
            all_MaxQ.append(max_queue_length)
            all_Wq.append(avg_waiting_time)

        # Store results with confidence intervals
        results["num_stations"].append(num_stations)
        results["mean_wait_time"].append(np.mean(all_Wq))
        results["std_err_wait_time"].append(sts.sem(all_Wq) * 1.96 if␣
 ↪len(all_Wq) > 1 else 0)  # 95% CI
        results["mean_queue_length"].append(np.mean(all_Lq))
        results["std_err_queue_length"].append(sts.sem(all_Lq) * 1.96 if␣
 ↪len(all_Lq) > 1 else 0)
        results["max_queue_length"].append(np.mean(all_MaxQ))

    return results
num_stations_list = np.arange(1, 21)  # Testing 1 to 20 service stations


# Time settings
warmup = 2 * 3600  # 2-hour warm-up to stabilize queues
run_time = 22 * 3600  # 22-hour operational time

# Run the experiment
empirical_results = run_experiment_stations(num_stations_list, arrival_rate,␣
 ↪service_dist, warmup, run_time, num_trials=10)

# Generate Plots with Confidence Intervals

fig, axes = plt.subplots(1, 3, figsize=(15,5))
```

```python
# Convert results to NumPy arrays
c_vals = np.array(empirical_results["num_stations"])
mean_Wq = np.array(empirical_results["mean_wait_time"])
std_Wq = np.array(empirical_results["std_err_wait_time"])
mean_Lq = np.array(empirical_results["mean_queue_length"])
std_Lq = np.array(empirical_results["std_err_queue_length"])
mean_MaxQ = np.array(empirical_results["max_queue_length"])

# Average Waiting Time
axes[0].errorbar(
    c_vals, mean_Wq, yerr=std_Wq,
    fmt='ro--', capsize=5, label="95% CI"
)
axes[0].set_xlabel("# of Service Stations")
axes[0].set_ylabel("Average Waiting Time (s)")
axes[0].set_title("Empirical Avg Waiting Time vs. # of Stations")
axes[0].grid(True)
axes[0].legend()

# Average Queue Length
axes[1].errorbar(
    c_vals, mean_Lq, yerr=std_Lq,
    fmt='bo--', capsize=5, label="95% CI"
)
axes[1].set_xlabel("# of Service Stations")
axes[1].set_ylabel("Average Queue Length")
axes[1].set_title("Empirical Avg Queue Length vs. # of Stations")
axes[1].grid(True)
axes[1].legend()

# Maximum Queue Length
axes[2].plot(
    c_vals, mean_MaxQ, 'go--', label="Max Queue Length"
)
axes[2].set_xlabel("# of Service Stations")
axes[2].set_ylabel("Max Queue Length")
axes[2].set_title("Empirical Max Queue Length vs. # of Stations")
axes[2].grid(True)
axes[2].legend()

plt.suptitle("Empirical Simulation Results (No Officer)")
plt.tight_layout()
plt.show()
```

```
Simulating different service station counts:   0%|          | 0/20 [00:00<?, ?it/s]

Skipping 1 stations ( =5.01   1, unstable).
Skipping 2 stations ( =2.50   1, unstable).
```

```
Skipping 3 stations ( =1.67   1, unstable).
Skipping 4 stations ( =1.25   1, unstable).
Skipping 5 stations ( =1.00   1, unstable).

Simulating different service station counts: 100%| | 20/20 [00:29<00:00,  1.45s/
```



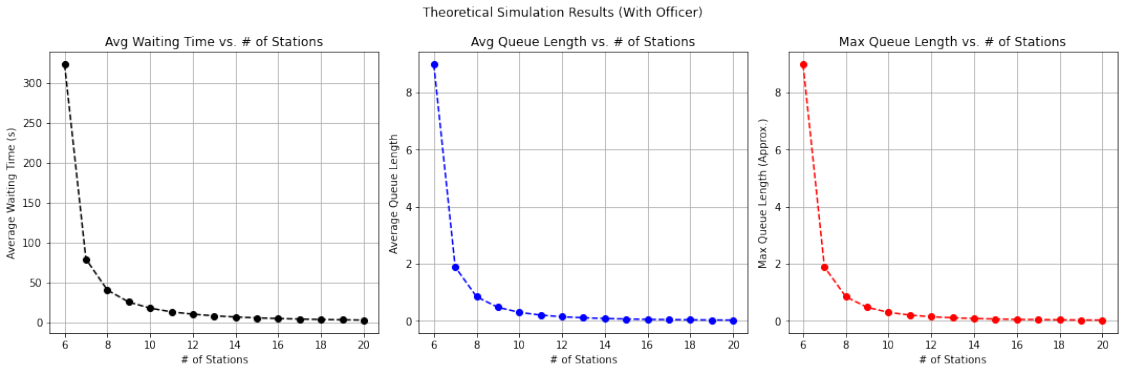
5.0.3 Theoretical - With Officer

Cell 16

[252]:
```python
# Theoretical M/G/1 model calculations (with officer considerations)
def mg1_with_officer_theoretical(num_service_stations_list, arrival_rate, E_Teff, Var_Teff):
    """
    Computes M/G/1 theoretical values for waiting time and queue length,
    considering the additional screening with the officer.
    """
    c_s_sq = Var_Teff / (E_Teff**2)
    theoretical_results = {"num_stations": [], "Wq": [], "Lq": []}

    for num_stations in num_service_stations_list:
        rho = (arrival_rate / num_stations) * E_Teff
        if rho >= 1.0:
            theoretical_results["Wq"].append(np.inf)
            theoretical_results["Lq"].append(np.inf)
        else:
            Wq = (rho**2 / (2 * (1 - rho))) * (1 + c_s_sq) * E_Teff
            Lq = Wq * arrival_rate / num_stations
            theoretical_results["Wq"].append(Wq)
            theoretical_results["Lq"].append(Lq)
        theoretical_results["num_stations"].append(num_stations)

    return theoretical_results

# Given parameters
```

```
num_service_stations_list = np.arange(1, 21)  # Testing from 1 to 20 stations

# Effective service time stats (considering officer delays)
p_extra = 0.03  # Probability of needing additional screening
E_T_basic = 30.0  # Mean basic service time (seconds)
Var_T_basic = 10.0**2  # Variance of basic service time
E_T_extra = 120.0  # Mean extra screening time (seconds)
Var_T_extra = 120.0**2  # Variance of extra screening time

# Effective service time calculations
E_Teff = (1 - p_extra) * E_T_basic + p_extra * (E_T_basic + E_T_extra)
Var_Teff = (1 - p_extra) * Var_T_basic + p_extra * (Var_T_basic + Var_T_extra)

# Compute theoretical predictions
theoretical_with_officer =␣
 ↪mg1_with_officer_theoretical(num_service_stations_list, arrival_rate,␣
 ↪E_Teff, Var_Teff)

# Plot theoretical results
fig, axes = plt.subplots(1, 3, figsize=(15,5))

# 1) Average Waiting Time (Wq)
axes[0].plot(theoretical_with_officer["num_stations"],␣
 ↪theoretical_with_officer["Wq"], 'ko--')
axes[0].set_xlabel("# of Stations")
axes[0].set_ylabel("Average Waiting Time (s)")
axes[0].set_title("Avg Waiting Time vs. # of Stations")
axes[0].grid(True)

# 2) Average Queue Length (Lq)
axes[1].plot(theoretical_with_officer["num_stations"],␣
 ↪theoretical_with_officer["Lq"], 'bo--')
axes[1].set_xlabel("# of Stations")
axes[1].set_ylabel("Average Queue Length")
axes[1].set_title("Avg Queue Length vs. # of Stations")
axes[1].grid(True)

# 3) Approximate Max Queue Length (using Lq as a proxy)
axes[2].plot(theoretical_with_officer["num_stations"],␣
 ↪theoretical_with_officer["Lq"], 'ro--')
axes[2].set_xlabel("# of Stations")
axes[2].set_ylabel("Max Queue Length (Approx.)")
axes[2].set_title("Max Queue Length vs. # of Stations")
axes[2].grid(True)

plt.suptitle("Theoretical Simulation Results (With Officer)")
plt.tight_layout()
```

```
plt.show()
```



5.0.4 Emperical - With Officer

Cell 17

[247]:
```
##############################################################################
# 1) Function: Run Empirical Simulations with Officer
##############################################################################

def run_experiment_stations_officer(num_stations_list, arrival_rate,␣
↪service_dist, additional_service_dist, p_extra, warmup, run_time,␣
↪num_trials=10):
    """
    Runs multiple replications of the airport security simulation with a single␣
↪officer for additional screening.
    Returns empirical means and 95% confidence intervals for:
        - Average waiting time Wq
        - Average queue length Lq
        - Maximum observed queue length
    """
    results = {
        "num_stations": [],
        "mean_wait_time": [],
        "std_err_wait_time": [],
        "mean_queue_length": [],
        "std_err_queue_length": [],
        "max_queue_length": [],
    }

    for num_stations in tqdm(num_stations_list, desc="Simulating different␣
↪service station counts (With Officer)"):
        all_Wq, all_Lq, all_MaxQ = [], [], []
```

```
        # Compute utilization  (should be < 1 for stability)
        E_T = service_dist.mean()  # Expected basic service time
        E_T2 = additional_service_dist.mean()  # Expected additional screening␣
↪time
        E_T_eff = E_T + p_extra * E_T2  # Effective service time
        rho = (arrival_rate / num_stations) * E_T_eff  # Utilization factor per␣
↪station
        if rho >= 1:
            print(f"Skipping {num_stations} stations ( ={rho:.2f}  1, unstable).
↪")
            continue  # Skip unstable cases

        for _ in range(num_trials):
            # Adjust arrival rate per station to avoid overload
            per_station_lambda = arrival_rate / num_stations
            arrival_dist = sts.expon(scale=1.0 / per_station_lambda)

            # Run the simulation for (warmup + steady-state time)
            total_time = warmup + run_time
            airport, _ = run_simulation_with_officer(arrival_dist,␣
↪service_dist, additional_service_dist, total_time, p_extra)

            # Extract queue data
            time_array = np.array(airport.queue.time_history)
            length_array = np.array(airport.queue.queue_length_history)

            # Filter only steady-state data (after warm-up period)
            length_array = length_array[time_array >= warmup]
            travelers = [rec for rec in airport.queue.traveler_records if␣
↪rec['arrival_t'] >= warmup and rec['start_t'] is not None]

            # Compute Metrics
            avg_queue_length = np.mean(length_array) if len(length_array) > 0␣
↪else 0
            max_queue_length = np.max(length_array) if len(length_array) > 0␣
↪else 0
            avg_waiting_time = np.mean([rec['start_t'] - rec['arrival_t'] for␣
↪rec in travelers]) if travelers else 0

            all_Lq.append(avg_queue_length)
            all_MaxQ.append(max_queue_length)
            all_Wq.append(avg_waiting_time)

        # Store results with confidence intervals
        results["num_stations"].append(num_stations)
        results["mean_wait_time"].append(np.mean(all_Wq))
```

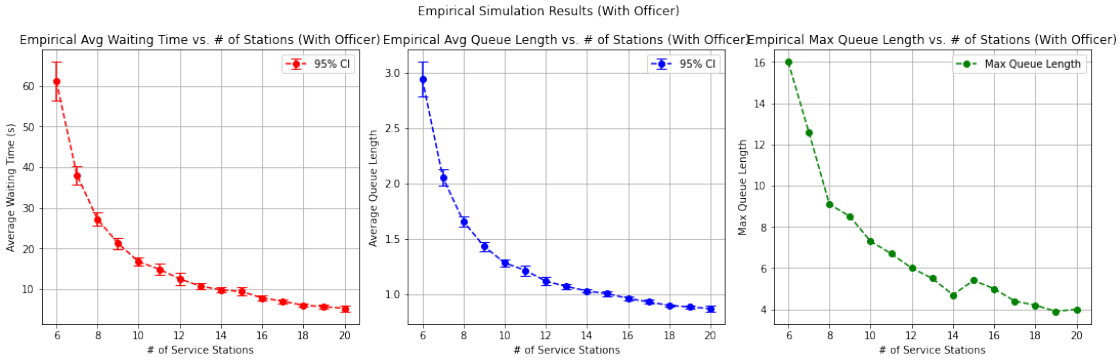
```
        results["std_err_wait_time"].append(sts.sem(all_Wq) * 1.96 if␣
 ↪len(all_Wq) > 1 else 0)  # 95% CI
        results["mean_queue_length"].append(np.mean(all_Lq))
        results["std_err_queue_length"].append(sts.sem(all_Lq) * 1.96 if␣
 ↪len(all_Lq) > 1 else 0)
        results["max_queue_length"].append(np.mean(all_MaxQ))

    return results

################################################################################
# 2) Run Experiment with Warm-up and 8-Hour Steady-State Period (With Officer)
################################################################################

# Truncated normal for Additional Screening (Handled by Single Officer)
mu_additional, sigma_additional = 120.0, 120.0  # 2 min ± 2 min
a2, b2 = (0 - mu_additional) / sigma_additional, np.inf
additional_service_dist = sts.truncnorm(a2, b2, loc=mu_additional,␣
 ↪scale=sigma_additional)

# Probability that a traveler needs additional screening
p_extra = 0.03

# Time settings
warmup = 2 * 3600
run_time = 22 * 3600

# Run the experiment
empirical_results_officer = run_experiment_stations_officer(num_stations_list,␣
 ↪arrival_rate, service_dist, additional_service_dist, p_extra, warmup,␣
 ↪run_time, num_trials=10)

################################################################################
# 3) Generate Plots with Confidence Intervals
################################################################################

fig, axes = plt.subplots(1, 3, figsize=(15,5))

# Convert results to NumPy arrays
c_vals = np.array(empirical_results_officer["num_stations"])
mean_Wq = np.array(empirical_results_officer["mean_wait_time"])
std_Wq = np.array(empirical_results_officer["std_err_wait_time"])
mean_Lq = np.array(empirical_results_officer["mean_queue_length"])
std_Lq = np.array(empirical_results_officer["std_err_queue_length"])
mean_MaxQ = np.array(empirical_results_officer["max_queue_length"])

# 1) Average Waiting Time
axes[0].errorbar(
```

```
    c_vals, mean_Wq, yerr=std_Wq,
    fmt='ro--', capsize=5, label="95% CI"
)
axes[0].set_xlabel("# of Service Stations")
axes[0].set_ylabel("Average Waiting Time (s)")
axes[0].set_title("Empirical Avg Waiting Time vs. # of Stations (With Officer)")
axes[0].grid(True)
axes[0].legend()

# 2) Average Queue Length
axes[1].errorbar(
    c_vals, mean_Lq, yerr=std_Lq,
    fmt='bo--', capsize=5, label="95% CI"
)
axes[1].set_xlabel("# of Service Stations")
axes[1].set_ylabel("Average Queue Length")
axes[1].set_title("Empirical Avg Queue Length vs. # of Stations (With Officer)")
axes[1].grid(True)
axes[1].legend()

# 3) Maximum Queue Length
axes[2].plot(
    c_vals, mean_MaxQ, 'go--', label="Max Queue Length"
)
axes[2].set_xlabel("# of Service Stations")
axes[2].set_ylabel("Max Queue Length")
axes[2].set_title("Empirical Max Queue Length vs. # of Stations (With Officer)")
axes[2].grid(True)
axes[2].legend()

plt.suptitle("Empirical Simulation Results (With Officer)")
plt.tight_layout()
plt.show()
```

```
Simulating different service station counts (With Officer):   0%|          | 0/20 [00:00<
```

```
Skipping 1 stations ( =5.78   1, unstable).
Skipping 2 stations ( =2.89   1, unstable).
Skipping 3 stations ( =1.93   1, unstable).
Skipping 4 stations ( =1.44   1, unstable).
Skipping 5 stations ( =1.16   1, unstable).
```

```
Simulating different service station counts (With Officer): 100%|| 20/20 [00:31
```


Empirical Simulation Results (With Officer)
Empirical Avg Waiting Time vs. # of Stations (With Officer)
Average Waiting Time (s)
# of Service Stations
95% CI
Empirical Avg Queue Length vs. # of Stations (With Officer)
Average Queue Length
# of Service Stations
95% CI
Empirical Max Queue Length vs. # of Stations (With Officer)
Max Queue Length
# of Service Stations
Max Queue Length