OPTIMIZING BED ALLOCATION IN HOSPITALS: AN IN-DEPTH ANALYSIS USING QUEUEING THEORY, SIMULATION, AND HEURISTIC METHODS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Hengtao Zhao

December 2024

ABSTRACT

Excessive wait times for admission to inpatient wards can lead to overcrowding in the departments where patients await admission. This issue is particularly concerning in emergency departments (ED) as it can prolong overall inpatient stays, higher mortality rates, and other undesirable events due to extended wait times [1]. To gain insights into reducing these wait times and provide effective management strategies for hospital administrators, we conducted a real-world study examining varying wait times across different service departments within the hospital. This research focuses on wait times, specifically examining the average time patients wait in line until admission to a bed in the steady-state (nonzero wait time) for each service department. We aim to optimize bed allocation in the hospital to minimize total wait times across all departments using a heuristic method.

In this research, queueing theory and simulation modeling were employed to analyze the hospital's wait times. Our findings revealed that: (1) the metrics results of simulation models for dynamic arrival rates with minimal variations (within 10%) closely matched the results of queueing theory with static arrival rates, and (2) the optimal bed allocation solutions from both methods using the simulated annealing algorithm were not exactly identical.

This study demonstrates that both queueing theory and simulation models are effective in capturing inpatient flow and producing comparable metrics for dynamic arrival rates with minimal variation. By understanding these methodologies, healthcare organizations can determine which approach to utilize in similar real-life scenarios, based on their desired level of analysis and computational expedience.

BIOGRAPHICAL SKETCH

Hengtao Zhao is a Master of Science student in the Systems Engineering Department at Cornell University. Born in China, he earned his undergraduate degree from Chongqing University of Technology. He later obtained a Master of Science in Industrial and Systems Engineering at Rutgers University. His research interests include queueing theory, simulation, machine learning, and mathematical programming. He is passionate about leveraging these methodologies to address complex real-world challenges in healthcare systems and manufacturing.

Dedicated to my parents, Suixiao Zhao and Ningguo He.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Professor Linda K. Nozick, for her unwavering support, guidance, and encouragement throughout my time at Cornell University. Her patience and kindness have been a source of inspiration, and her insightful advice and constructive feedback have been invaluable in shaping both my research and my academic journey. Professor Nozick's dedication to her students and her passion for advancing knowledge have profoundly influenced my approach to problem-solving and learning. I am truly fortunate to have had the opportunity to learn from her and benefit from her expertise and mentorship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the realm of hospital operations, the intricate interplay between resource availability (beds, nurses, physicians) and patient wait times for inpatient admission has long been a focal point for researchers. Prolonged delays in securing a hospital bed can lead to a cascade of adverse events, jeopardizing patient care. Patients awaiting admission may find themselves in suboptimal conditions, unable to receive essential treatments, such as urgent antibiotics, that could mitigate their symptoms or prevent complications.

Studies in [2, 1] underscored the gravity of this issue, revealing that patients who waited more than six hours after their admission decisions were made were at increased risk of experiencing extended inpatient stays and higher mortality rates. Moreover, the presence of patients occupying waiting areas can exacerbate overcrowding, hindering the treatment of new arrivals.

The emergency department (ED) serves as the primary gateway for inpatient admissions, and the phenomenon of ”ED boarding,” wherein patients are compelled to await for inpatient beds within the ED, has been recognized as a significant factor contributing to ED overcrowding [3, 4]. Consequently, numerous studies have delved into the intricacies of patient flow within EDs. Recent research has highlighted the steady increase in patient volume seeking emergency care, leading to overcrowding and prolonged wait times[5].

Among the critical resources within hospitals, inpatient beds play a pivotal role [6]. Understanding the dynamics of how ED patients transition to inpatient beds is essential for mitigating wait times, averting chaos, and enhancing patient satisfaction.

1.1 Motivation and research questions

In the real world, there are numerous emergency hospitals dedicated to providing immediate care for patients with urgent medical needs. These institutions, such as Massachusetts General Hospital and Johns Hopkins Hospital, typically possess well-equipped emergency departments and specialized units for trauma, critical care, and other emergency services. Within these hospitals, inpatients are often classified by medical specialties, including Surgery, Cardiology, Orthopedic, Oncology, and Neurology, among others (the categorization varies by hospital).

By investigating ED boarding in emergency hospitals and developing models to optimize inpatient flow, we aim to gain valuable insights that can be applied to general hospitals. Our study seeks to create models that capture the dynamics of various inpatient flows within emergency hospitals and explore strategies for allocating existing beds to minimize overall average wait times across different specialties. While the models presented in this paper are based on an empirical study at an emergency hospital, we believe that the modeling framework can serve as a foundation for further study in general hospitals and can be extended to those with similar configurations at their EDs that have dedicated wards for sources other than the ED.

1.2 Contributions

This study makes two primary contributions to the field of hospital operations modeling.

Performance Measure Selection: Regarding performance measure selection, traditional analyses have focused on average wait time when analyzing patient flow in the ED. However, average wait time can be misleading because it includes patients who do

not wait, thereby diluting the average. To address this limitation, we propose using the average time a patient waits in line until admitted to a bed in the steady-state (also known as nonzero wait time) for each service department at the hospital. By excluding those who do not wait, this measure offers a clearer picture of wait times and a more accurate representation of the waiting experience.

Model Comparison: Regarding model comparison, previous studies have tended to analyze performance measures from either a queueing theory or simulation perspective. In this study, we compared various metrics, including service times, queue lengths, and utilization, from both perspectives. By comparing the mean values of these metrics from queueing theory and the 95% confidence intervals from simulation modeling, we were able to draw valuable conclusions about the most suitable method for real-world case analysis.

Little research has developed models that analyze nonzero wait time as the primary performance measure. Therefore, the investigation in this paper offers a novel perspective on ED boarding studies. Additionally, we compared the main metrics using both queueing theory with static arrival rates and simulation models with dynamic arrival rates. This approach provides valuable insights into metric analysis and helps determine the most suitable method for real-world case analysis.

1.3 Outline

The remainder of this paper is organized as follows: Section 2 provides a comprehensive literature review on inpatient flow and hospital operations. Section 3 introduces queueing theory and simulation modeling as tools for formulating inpatient flow. It presents formulas for calculating metrics such as average wait time, queue length, throughput, and

utilization using queueing theory. Additionally, the algorithm for obtaining these metrics through simulation is described. Furthermore, the simulated annealing algorithm is introduced as a method for optimizing inpatient bed allocation. Section 4 conducts a case study of an emergency hospital with eight medical specialties. The metrics calculated using both queueing theory and simulation are compared to draw conclusions. The simulated annealing algorithm is applied to identify an optimal bed allocation strategy that minimizes the total mean nonzero wait times. Finally, Section 5 discusses the findings of this investigation, highlights potential research directions, and emphasizes the contributions of this study.

CHAPTER 2

LITERATURE REVIEW

Healthcare systems, particularly hospitals, are inherently complex and closely linked to public welfare[7]. Over recent years, the increasing number of patients has exacerbated inefficiencies in hospital processes, driving up costs and straining resources[5]. Given the significance of healthcare operations, many studies have focused on optimizing these processes to improve efficiency. Since healthcare systems often involve complex processes like patient flow and resource management, they are a prime application area for operations research techniques. Queueing theory, for example, has been widely used to model and optimize wait times and resource allocation in hospital settings[8]. This section reviews relevant academic research on hospital operations, the methods used, and the findings, highlighting potential problems and offering insights into possible solutions.

Extensive research has been conducted on hospital patient flow, underscoring the importance of effective patient flow management. Improving patient flow can significantly enhance care quality, patient satisfaction, and hospital efficiency[7, 9, 10]. By optimizing patient flow, hospitals can better allocate resources, reduce staff strain, and minimize costs. Discrete-event simulation (DES) and queueing theory are the most commonly used approaches in this field.

Emergency departments (ED) are often the focus of such studies, as they serve as major entry points for hospital admissions and are a key source of overcrowding. Several studies have investigated the factors influencing patient length of stay (LOS) in EDs. For instance, [11]identified service-related factors—such as the complexity of care, initial ward designation, and ward transfers—that significantly prolong patient stays in the ED, offering practical insights into improving patient flow.

Queueing models and simulation-based approaches have also been widely applied

to study ED patient flow. [12]used queueing theory to analyze patient flow within an ED, providing reasonably accurate performance evaluations of the system. Similarly, [13]combined DES with agent-based simulation and multi-attribute decision-making methods to improve patient throughput in EDs.

In addition to patient flow, hospital operations—such as budget management, staffing, quality of care, and service availability—are critical to enhancing hospital efficiency and patient satisfaction. Studies on resource allocation, including ward nurse staffing[14, 15, 16, 17] and bed allocation[18, 19], have been particularly intensive. For example, [20] applied queueing theory to improve quality of care, validating the queueing theory's effectiveness in hospital operations. [17] utilized simulation modeling to evaluate different staffing scenarios and their impact on patient outcomes and cost-effectiveness, helping to mitigate the challenges of nurse shortages.

Bed allocation studies have also benefited from queueing theory and simulation modeling. For instance, [19] employed a DES model to predict bed occupancy for COVID-19 patients, considering factors such as patient severity and inter-hospital transfers. Similarly, [21] used queueing theory and simulation model to analyze bed allocation by evaluating arrival rates, service times, and bed availability, offering performance metrics such as patient wait times and resource utilization to assess the system's effectiveness.

In addition to queueing theory and simulation models, other methodologies—such as stochastic network models, dynamic programming (DP), and machine learning (ML)—have also been applied to hospital processes[22, 23, 24, 25]. For instance, [22] used stochastic network models to examine how the time of day affects ED wait times for patients requiring inpatient beds, offering valuable insights for reducing admission delays. [23] applied dynamic programming to optimize clinical decision-making, such as determining the best treatment policies. Meanwhile, [24, 25] investigated machine learning techniques for predicting emergency patient admissions and improving scheduling

efficiency.

While these alternative approaches have proven effective, queueing theory and DES remain the most widely used due to their adaptability and effectiveness in addressing the complexities of real-world hospital operations. However, a gap in the research still exists, as few studies have applied multiple methodologies to the same problem. Integrating different approaches could yield more robust solutions for tackling the challenges of hospital operations.

Additionally, after [26] demonstrated that measuring ED boarding times—i.e., the wait time for ED patients awaiting admission to inpatient beds—helps identify inefficiencies in the ED, most subsequent studies adopted wait times, particularly average wait times, as key performance metrics when doing analysis. However, using average wait time can be misleading, as it includes patients who experience no wait, thus skewing the result. To overcome this limitation, we propose using the average nonzero wait time as a more suitable alternative to the average wait time. This approach only considers patients who actually experienced a wait.This metric provides a clearer, more accurate representation of the patient waiting experience by excluding those who are admitted immediately.

Building on this, our research compares the use of queueing theory and DES in analyzing hospital bed allocation, specifically targeting the minimization of patient nonzero wait times. Our goal is to determine which method is more suitable for real-world applications, with a particular focus on optimizing bed allocation to improve patient outcomes.

The following sections delve into a case study of a hospital focused on emergency services with multiple medical specialties. We will explain in detail how both queueing theory and DES can be used to model patient flow through a network of queues, comparing their performance and providing insights into improving hospital operations.

CHAPTER 3

MODELING

3.1 Introduction

In this chapter, we examined a hospital system and optimized the allocation of beds across various medical specialties using different methods. We employed two robust analytical methods, namely simulation and queueing theory, to delve into the system's dynamics and enhance its performance where possible. As outlined in Chapter 1, the primary aim of this study is to boost the system's efficiency by minimizing the total average nonzero wait time for patients across all medical specialties. This paper utilized proven heuristic methods to identify the optimal bed configuration that would achieve the primary goal. Specifically, we explored the application of the simulated annealing algorithm to accomplish this objective.

3.2 Hospital Overview

This section provided an overview of the hospital and its inpatient flow, focusing on emergency department (ED) patients [1]. The hospital comprises various service departments, each dedicated to serving a specific medical specialty, and a pool of general beds for inpatient care.

A typical scenario involves an ED patient unfolds as follows. Take, for example, a patient with a cardiac condition. After initial treatment, including potential surgery, the

[1] this investigation centers on hospitals specializing in emergency care, where the occupancy of general beds by patients not admitted through the emergency department is negligible. As a result, the subsequent discussion will exclusively address the occupancy of general beds by emergency department patients.

patient may be transferred to a waiting area for observation. Some patients may be discharged directly from the ED, while others will eventually be admitted to the general wards[2][20]. Those admitted to the general wards and staying in a bed will finally be discharged. During this process, there is a wait time before the inpatients can be admitted to the general wards. This study takes a particular interest in this wait time, especially when it is nonzero, and will analyze it thoroughly.

3.3 Schematic of the Studied Hospital

Figure 3.1 presents a schematic representation of a hospital system that accommodates inpatients into the general ward and subsequently discharges them. The hospital system is comprised of 'n' individual service departments (Stream 1 to n), each corresponding to a different medical specialty. Each service department has a specific server configuration and specializes in treating a particular type of patients. This hospital system is essentially a network of queuing systems, detailed as follows:

Service Departments (Stream 1 to n):

1. Patients arrive at each service department according to a Poisson process with a time-varying arrival rate.
2. Each service department has a fixed number of servers (beds), each with an exponential service time distribution.
3. When a new patient arrives at a service department and all beds are occupied, they must wait in a queue before being admitted to a ward and assigned a bed.
4. Patients in the patient stream are assigned beds on a first-in-first-out (FIFO) policy.

[2]This paper excludes readmission cases, focusing on inpatients admitted into general wards and then discharged after receiving service. This is maintained by collecting only relevant data.

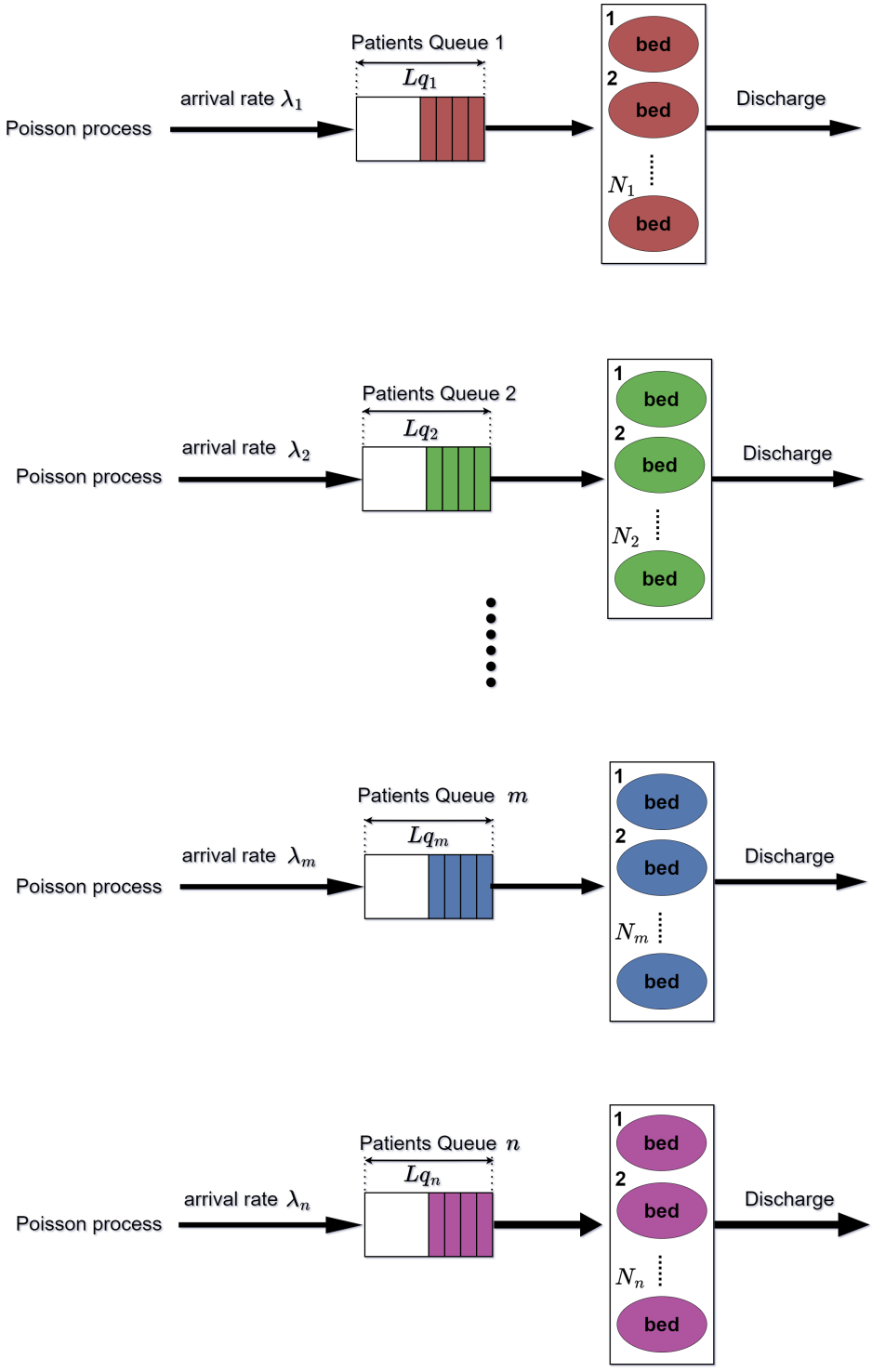


Figure 3.1: Patient Flow Network

5. Upon completing their treatment, patients are discharged, freeing up beds for incoming patients.

Table 3.1 enumerates the parameters utilized in this paper for the analysis of the hospital system[3].

service department	**Medical Specialty**	**Arrival Rate (persons per hour)**	**Number of Beds**	**Service Rate (persons per hour)**
Stream 1	Cardiology	λ_1	k_1	μ_1
Stream 2	Orthopedic	λ_2	k_2	μ_2
...	...	...	...	...
Stream n	Oncology	λ_n	k_n	μ_n

Table 3.1: Model Parameters

Remarks:

1. In service departments, patient arrival processes and service processes are assumed to follow Poisson processes, as described in [22].
2. The initial number of servers (beds) in these service departments is based on empirical data.
3. The specific medical specialties offered vary across hospitals but typically include Cardiology, Orthopedic, Surgery, Oncology, General Medicine, Neurology, Renal Disease, Respiratory, Gastroenterology-Endocrine, and others.
4. Throughout this paper, the terms ”customer,” ”arrival,” and ”patient” are used interchangeably, as are ”bed” and ”server.”

[3]The derivation of these parameters is not the focus of this paper, so we have omitted this step and used parameters from other sources.

3.4 Queueing Theory

Building upon the hospital overview provided earlier, we sought to create models that could describe, analyze, and address the challenges faced by this healthcare facility. Recognizing the applicability of queueing theory, we assumed static arrival rates for each service department and employed M/M/k queue analysis to capture the complexities of the hospital's operations.

Given the M/M/k nature of the processes within each service department, we adopted this system as the basis for our theoretical analysis in this paper. Specifically, within the framework of queueing theory, we established the following statistical parameters for these M/M/k systems:

- $\rho =:$ *utilization*, the average proportion of time in which each of the servers is occupied in a steady-state (the long-term behavior of the system where the probabilities of being in each state remain constant over time). Its value must be less than or equal to 1;
- $\pi_0 =:$ the probability that there are no customers in the system in the steady-state;
- $P_Q =:$ the probability that a new customer, upon arrival, finds all the beds occupied and must therefore wait in line in the steady- state;
- $E(W) =:$ the average time a customer spends in the wait line for service in the steady-state;
- $E(W|W > 0) =:$ the average time a customer has to wait in line until receiving service in the steady-state;
- $L_q =:$ the average number of customers in wait line wait for service in the steady-state;

- L =: the average number of customers in the system (in wait line and being served) in the steady-state;
- τ =: the average time a customer spends being served in the steady-state;
- S =: the average delay per customer, also known as the sojourn time (wait time in queue plus service time) in the steady-state;
- T =: *throughput*, or departure rate, the average rate at which customers leave the system in the steady-state;

For an M/M/k system with arrival rate λ, service rate μ, and number of server k, the following steady-state statistics were derived using queueing theory and are presented in Table 3.2.

service department	ρ **(Utilization)**	π_0 **(Probability of zero patients)**	P_Q **(Probability of all servers being busy upon arrival)**	$E(W)$ **(Mean wait time)**	$E(W\|W>0)$ **(Mean wait by those who are blocked)**	L_q **(Mean queue length)**	L **(Mean system length)**	τ **(Mean service time)**	S **(Mean sojourn time)**	T **(Throughput)**
Formula	$\rho = \frac{\lambda}{k\mu}$	$\pi_0 = \left[\sum_{i=0}^{k-1} \frac{(k\rho)^i}{i!} + \frac{(k\rho)^k}{k!(1-\rho)}\right]^{-1}$	$P_Q = \frac{(k\rho)^k}{k!(1-\rho)}\pi_0$	$E(W) = \frac{P_Q}{(1-\rho)k\mu}$	$E(W\|W>0) = \frac{1}{(1-\rho)k\mu}$	$L_q = \frac{P_Q\rho}{(1-\rho)}$	$L = L_q + \frac{\lambda}{\mu}$	$\tau = \frac{1}{\mu}$	$S = E(W) + \tau$	$T = \lambda$

Table 3.2: Formulas for M/M/k system statistics

Using the parameters provided in Table 3.1, the following statistics for the hospital were derived and are presented in Table :

service department	ρ **(Utilization)**	π_0 **(Probability of zero patients)**	P_Q **(Probability of all servers being busy upon arrival)**	$E(W)$ **(Mean wait time)**	$E(W\|W>0)$ **(Mean wait by those who are blocked)**	L_q **(Mean queue length)**	L **(Mean sojourn length)**	τ **(Mean service time)**	S **(Mean system time)**	T **(Throughput)**
Stream 1	$\rho_1 = \frac{\lambda_1}{k_1\mu_1}$	$\pi_{01} = \left[\sum_{i=0}^{k_1-1} \frac{(k_1\rho_1)^i}{i!} + \frac{(k_1\rho_1)^{k_1}}{k_1!(1-\rho_1)}\right]^{-1}$	$P_{Q1} = \frac{(k_1\rho_1)^{k_1}}{k_1!(1-\rho_1)}\pi_{01}$	$E(W_1) = \frac{P_{Q1}}{(1-\rho_1)k_1\mu_1}$	$E(W_1\|W_1>0) = \frac{1}{(1-\rho_1)k_1\mu_1}$	$L_{q1} = \frac{P_{Q1}\rho_1}{1-\rho_1}$	$L_1 = L_{q1} + \frac{\lambda_1}{\mu_1}$	$\tau_1 = \frac{1}{\mu_1}$	$S_1 = E(W_1) + \tau_1$	$T_1 = \lambda_1$
Stream 2	$\rho_2 = \frac{\lambda_2}{k_2\mu_2}$	$\pi_{02} = \left[\sum_{i=0}^{k_2-1} \frac{(k_2\rho_2)^i}{i!} + \frac{(k_2\rho_2)^{k_2}}{k_2!(1-\rho_2)}\right]^{-1}$	$P_{Q2} = \frac{(k_2\rho_2)^{k_2}}{k_2!(1-\rho_2)}\pi_{02}$	$E(W_2) = \frac{P_{Q2}}{(1-\rho_2)k_2\mu_2}$	$E(W_2\|W_2>0) = \frac{1}{(1-\rho_2)k_2\mu_2}$	$L_{q2} = \frac{P_{Q2}\rho_2}{1-\rho_2}$	$L_2 = L_{q2} + \frac{\lambda_2}{\mu_2}$	$\tau_2 = \frac{1}{\mu_2}$	$S_2 = E(W_2) + \tau_2$	$T_2 = \lambda_2$
...	...	...	...	...	...	...	...	...	...	...
Stream n	$\rho_n = \frac{\lambda_n}{k_n\mu_n}$	$\pi_{0n} = \left[\sum_{i=0}^{k_n-1} \frac{(k_n\rho_n)^i}{i!} + \frac{(k_n\rho_n)^{k_n}}{k_n!(1-\rho_n)}\right]^{-1}$	$P_{Qn} = \frac{(k_n\rho_n)^{k_n}}{k_n!(1-\rho_n)}\pi_{0n}$	$E(W_n) = \frac{P_{Qn}}{(1-\rho_n)k_n\mu_n}$	$E(W_n\|W_n>0) = \frac{1}{(1-\rho_n)k_n\mu_n}$	$L_{qn} = \frac{P_{Qn}\rho_n}{1-\rho_n}$	$L_n = L_{qn} + \frac{\lambda_n}{\mu_n}$	$\tau_n = \frac{1}{\mu_n}$	$S_n = E(W_n) + \tau_n$	$T_n = \lambda_n$

Table 3.3: Statistics for Hospital Modeling

Remarks:

1. While the M/M/k queueing analysis assumes a static arrival rate, which may deviate from the real-world dynamics, it still serves as a valuable baseline because by simplifying the arrival rate to a static value, we can rapidly establish a foundational model, which offers a quick overview of hospital operation. The statistics of this model often provide a reasonable approximation to reality.
2. With these statistical computations in hand, we can focus on optimizing the hospital to achieve specific objectives, such as minimizing the overall average nonzero wait time.

While M/M/k queueing analysis provides an assessment of hospital operations with static arrival rate, it fails to capture the dynamic complexities of patient arrivals. Therefore, we extended the modeling to simulation which more accurately reflects the dynamic nature of patient arrivals. The following section detailed the simulation development process.

3.5 Simulation Analysis

A simulation model can replicate the dynamic nature of patient arrivals, enabling us to explore the intricacies of the hospital operations and understand its underlying mechanisms. By collecting data and deriving statistics from the simulation, we can mimic the behavior of the real system. We developed a discrete-event simulation model and provided a detailed description as follows.

3.5.1 Discrete-Event Simulation (DES)

In 1970, Schmidt and Taylor defined a system as a collection of entities, e.g., people or machines, that act and interact together toward the accomplishment of some logical end[27]. A system's state is a set of variables that captures the essential properties of the system, such as the number of customers in the system.[28]. These variables represent the status of the system and can changer. An event is an instantaneous occurrence that may change the state of the system[29], like a customer's arrival or departure. Accordingly, a discrete system is one for which the state variables change instantaneously at separate points in time.

A discrete-event simulation (DES) is a type of simulation model used to represent discrete system in the real-world. The hospital in this study is an example of a discrete system [30] because its state variables, such as the number of patients in a service department, only change when a patient arrives or departs. Consequently, the hospital can be simulated using DES, and its events are listed in Table3.4.

3.5.2 Simulation of a Multiple-Server Queueing System

The hospital can be modeled as a network of multiple M(t)/M/k queueing systems. To simplify our analysis, we focused on a single M(t)/M/k system as a representative example of each service department within the network.

In the DES model of a service department, the flowcharts for the arrival and departure events help ensure that the simulation model accurately represents the real-world system and behaves as expected [31][30]. Figure 3.2 illustrates the effect of the patient arrival events within a service department. First, the time of the next arrival in the future is generated and placed in the event list. Then a check is made to determine whether the

Event description	Event type
Arrival of a patient to a service department	1
Discharge of a patient from the service department after completing service	2
Stopping event, scheduled to occur at a preset time	3

Table 3.4: Hospital System Event Table

server is busy. If so, the number of customers in the queue is incremented by 1, and we ask whether the storage space allocated to hold the queue is already full. If the queue is already full, an error message is produced and the simulation is stopped; if there is still room in the queue, the arriving customer's time of arrival is put at the (new) end of the queue. (This queue-full check could be eliminated if using dynamic storage allocation in a programming language that supports this.) However, if the arriving customer finds the server idle, then this customer has a delay of 0, which is counted as a delay, and the number of customer delays completed is incremented by 1. The server must be made busy, and the time of departure from service of the arriving customer is scheduled into the event list.

The effect of the patient departure events is shown in detail in fig. 3.3. This routine is invoked when a service completion (and subsequent departure) occurs. If the departing customer leaves no other customers behind in queue, the server is idled and the departure event is eliminated from consideration, since the next event must be an arrival. However,

if one or more customers are left behind by the departing customer, the first customer in queue will leave the queue and enter service, so the queue length is reduced by 1, and the delay in queue of this customer is computed and registered in the appropriate statistical counter. The number of delayed patients is increased by 1, and a departure event for the customer now entering service is scheduled. Finally, the rest of the queue (if any) is advanced one place.

We used an M(t)/M/k queue as an example to illustrate a potential time path, or realization, of the queue. Figure 3.4 shows a potential path of an an M(t)/M/3 queue.

We initialized the simulation with a clock time of 0 and an empty queue ($L_q = 0$). Then we generated an interarrival time for the first patient. If a bed was available (in this case, bed 1 was selected randomly), the patient occupied it immediately and we generated their service time. Otherwise, the patient joined the queue. We repeated this process for subsequent patients, updating the queue length and wait times as needed.

When a patient's service was completed, they were discharged, freeing up a bed. If there were patients wait in the queue, the next patient was admitted. This process continued until the simulation reached the preset termination time, T.

Throughout the simulation, the queue length fluctuated based on the arrival of new patients and the availability of beds.

The algorithm outlined below represents a single M(t)/M/k system. The primary statistics collected within this system include the system's utilization and the probability of all beds being occupied. Additionally, we can gather other key metrics such as throughput, average nonzero wait time, and more when we conduct a real case study.

Having developed both the M/M/k queueing model with static arrival rates and the DES model with dynamic arrival rates, we then explored how to allocate beds to each

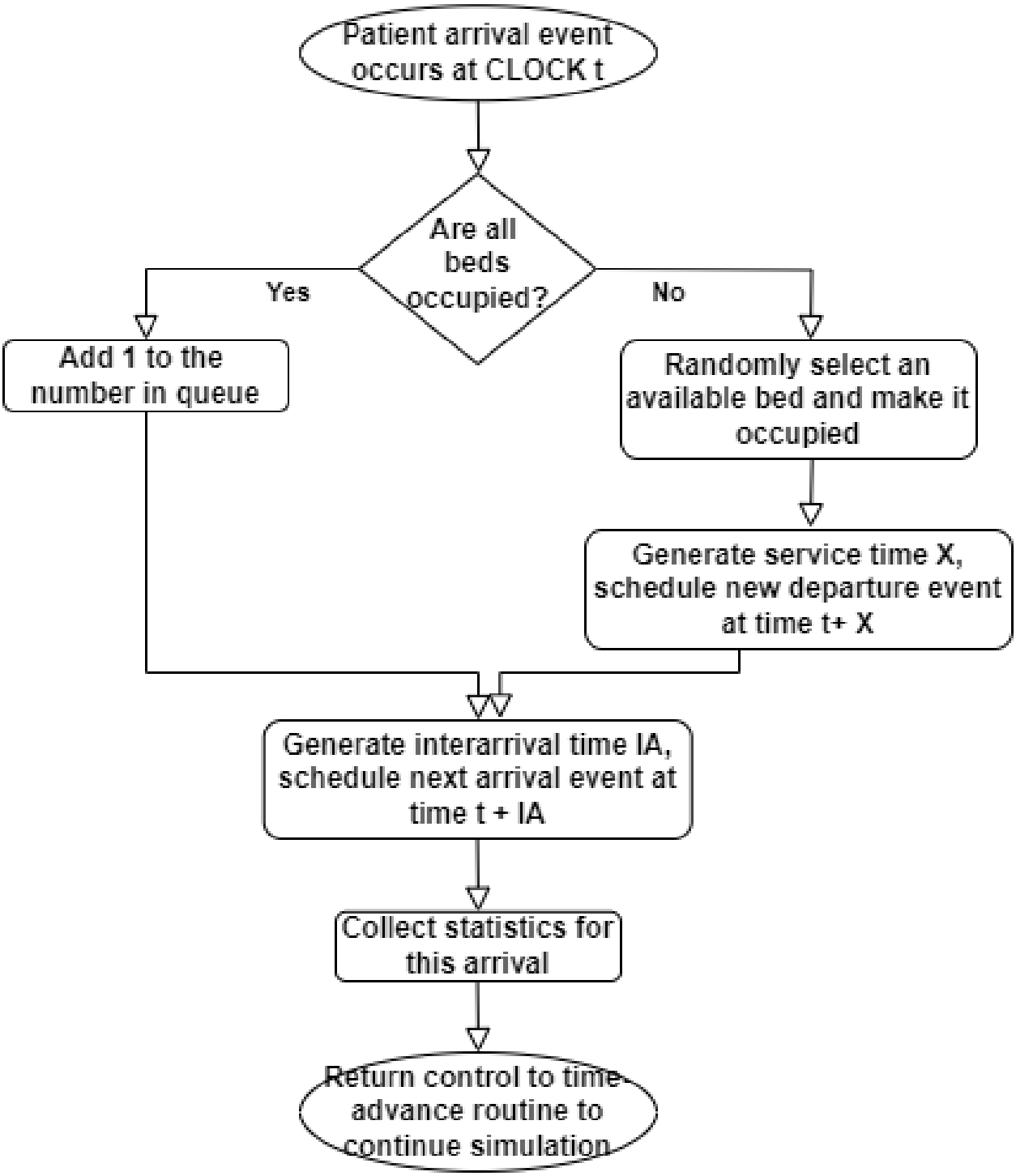


Figure 3.2: Flowchart for arrival routine, queueing model for a service department

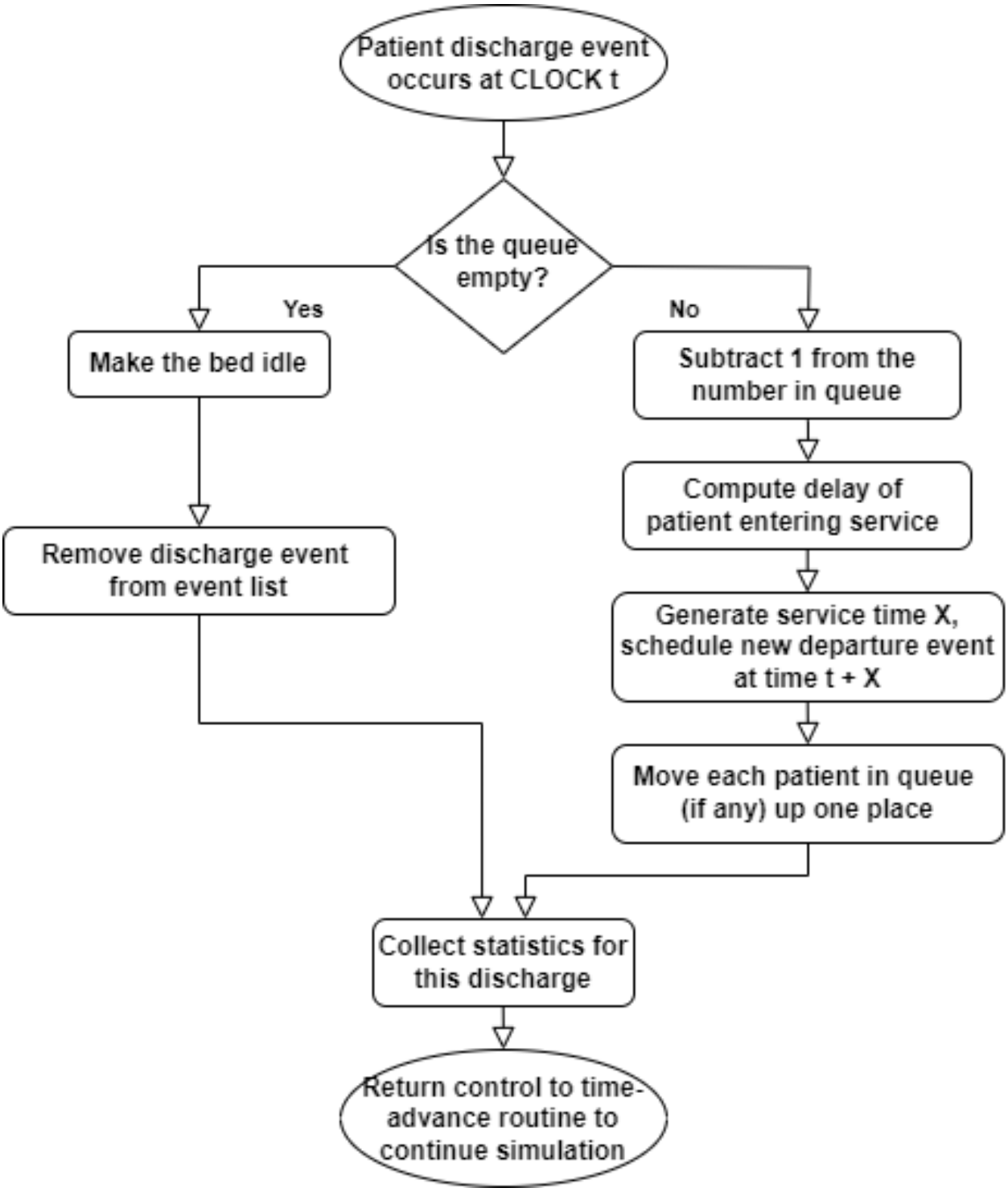


Figure 3.3: Flowchart for discharge routine, queueing model for a service department

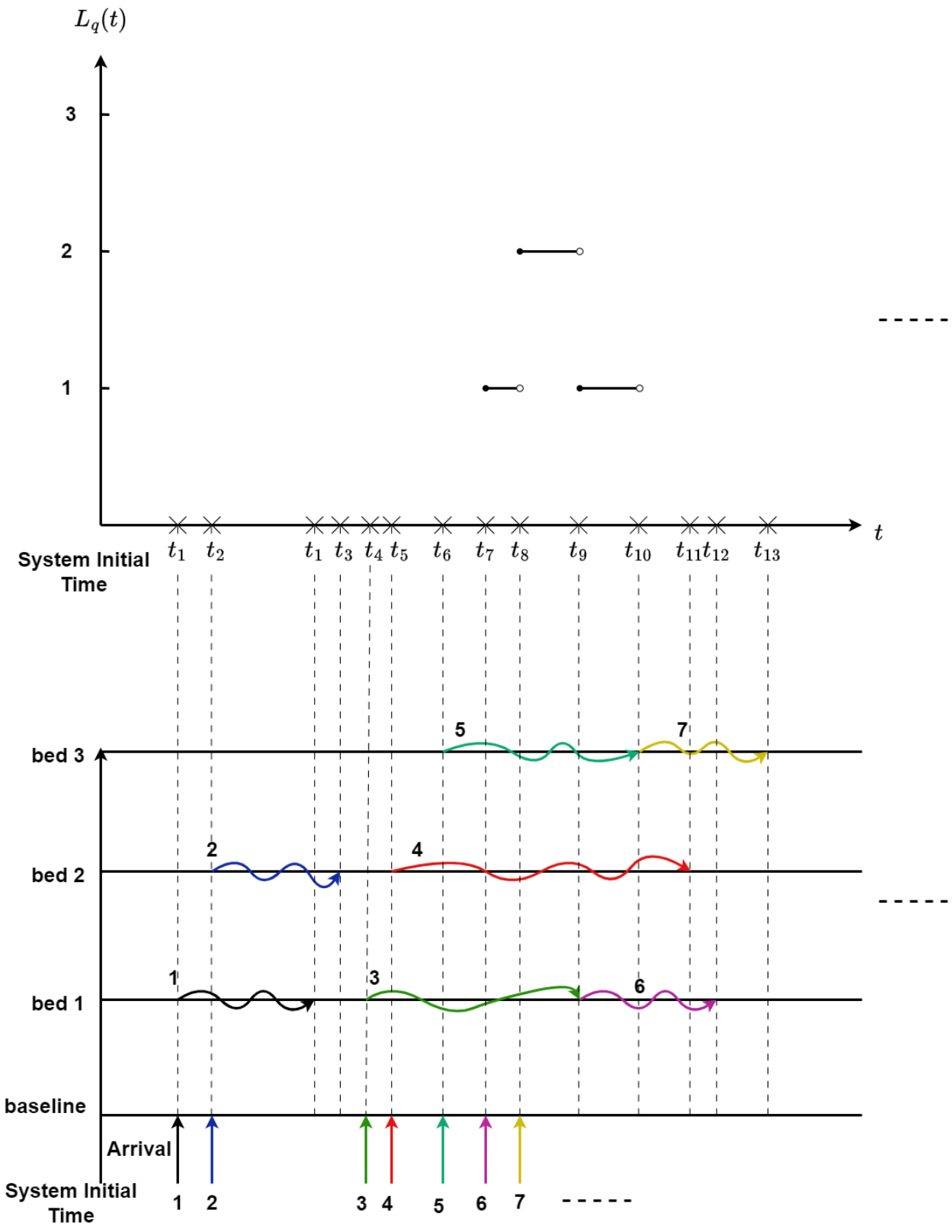


Figure 3.4: Patient Queue Animation

Algorithm 1 Simulate M(t)/M/k Queue

Initialize parameters: $\lambda(t)$, μ, k
Initialize simulation clock to 0
Initialize system state: empty queue, all servers idle
Schedule first arrival event, using accept-reject method with rate $\lambda(t)$
while simulation is running **do**
    Identify next event (arrival or departure)
    Advance simulation clock to time of next event
    if event is arrival **then**
        if bed is available **then**
            Assign patient to a bed
            Schedule departure event with rate μ
        else
            Enqueue the patient
        end if
        Schedule next arrival event, using accept-reject method with rate $\lambda(t)$
    else if event is departure **then**
        Remove patient from bed
        if queue is not empty **then**
            Assign the first patient in queue to a bed
            Schedule new departure event
        end if
    end if
    Collect statistics (such as waiting times, queue lengths, server utilization)
end while
End simulation
Calculate and report statistics

service department to minimize the total wait time across the service departments. The following section delved into this problem and presented a method to address it.

3.6 Optimization

Given that we have had a thorough understanding of hospital operations, it is natural to inquire about optimizing bed allocation to achieve specific goals at the management level. These goals can be complex, such as minimizing the total wait time across all patient streams, minimizing the total wait time for patients who experience delays (total nonzero

wait time across all patient streams), or maximizing overall utilization while maintaining reasonable wait times.

Minimizing the total weighted nonzero wait time across all patient streams is a common objective in healthcare. This approach takes into account the relative impact of different streams. For instance, the weighted mean nonzero wait time for a stream is calculated by multiplying the mean nonzero wait time for that stream by its arrival rate relative to the total arrival rate across all streams. This weighting ensures that patient streams with higher arrival rates have a greater impact on the overall nonzero wait time metric. This paper aims to achieve this goal by investigating how to allocate a fixed number of beds optimally across all streams.

3.6.1 Problem Formulation

The problem of minimizing the total weighted mean nonzero wait time for a hospital system can be formulated as a nonlinear integer programming model. The mathematical formulation for this problem is described as follows:

- **Objective Function:**

  The goal is to minimize the total weighted mean nonzero wait time across all patient streams. This is represented mathematically as:

$$\min . \sum_{i=1}^{N} \alpha_i * E(W_i | W_i > 0) \tag{3.1}$$

  Remarks:

  1. Each patient stream is represented by an M(t)/M/k queue system. Here, N is the number of the M/M/k queues, and $E(W_i|W_i > 0)$ is the mean nonzero wait time for the i^{th} M(t)/M/k queue.

2. There are different approaches to consider the weight of $E(W_i|W_i > 0)$ across all the service departments:

   (a) One approach is to weight them proportionally to the number of patients in that service department. The weight for the i^{th} M/M/k queue's $E(W_i|W_i > 0)$ can then be determined by:

      i. Its relative arrival rate or the percentage of population it impacts. This can be obtained by the fraction of population of all of the service departments.

      ii. The probability that a new patient arriving at the service department finds all beds occupied and must queue in a steady-state.

   (b) Another approach is to assign weights to the $E(W_i|W_i > 0)$ values across all service departments based on their relative importance. For instance, if we determine that the importance of all $E(W_i|W_i > 0)$ values is equal, we can assign them equal weights and calculate the weighted sum as follows:

$$1 \cdot E(W_1|W_1 > 0) + 1 \cdot E(W_2|W_2 > 0) + \ldots + 1 \cdot E(W_i|W_i > 0)$$

3. In this application, we assume that all patient streams have equal importance and urgency, and all patients are treated with uniform priority. To avoid potential issues with extremely small or large values of $E(W_i|W_i > 0)$, we have squared this term in the objective. Therefore, the objective becomes:

$$\min . \sum_{i=1}^{N} [E(W_i|W_i > 0)]^2 \tag{3.2}$$

- **Constraints:**

1. *Mean Nonzero wait Time Constraints:*

   Each M/M/k queue's mean nonzero wait time must satisfy predefined constraints:

$$E(W_i|W_i > 0) > 0, \quad i = 1, 2, ..., N \tag{3.3}$$

These constraints ensure that the mean wait time for patients who experience a wait in each M/M/k queue is greater than 0.

2. *Server Constraints:*

$$\sum_{i=1}^{N} k_i = n, \quad i = 1, 2, ..., N \tag{3.4}$$

$$0 \leq k_i \leq n, \quad \forall k_i \in integer \tag{3.5}$$

These constraints ensure that the total number of servers across all M/M/k queues equals the total number available, (n), while also keeping the number of the servers in each M/M/k queue within its own limits.

3. *Utilization Constraints:*

$$0 \leq \rho_i \leq 1, \quad i = 1, 2, ..., N \tag{3.6}$$

These constraints ensure that for every M/M/k queue, its utilization is maintained between 0 and 1.

Remark:

(a) $\rho_i \leq 1$ is necessary; otherwise, there is no feasible solution as each queue will never reach its steady-state.

After developing the mathematical model for the hospital system, we explored various solution approaches for this nonlinear integer programming problem in the discrete space. One intuitive approach is the Brute Force Method. This method involves systematically checking every possible combination of bed assignments across all patient streams to verify if each candidate satisfies the constraints until the correct solution is found. By comparing the computation results of each combination, we can eventually determine the optimal bed assignment for each medical specialty. However, this method is computationally expensive due to the large number of possible combinations, especially given the typically high number of beds in a hospital. Alternatively, heuristic methods have demonstrated their effectiveness in narrowing the search space for complex problems.

Therefore, we employed a heuristic approach to solve this nonlinear programming problem. Specifically, we applied simulated annealing algorithm to determine the optimal bed allocation for the hospital.

3.6.2 Simulated Annealing Algorithm (SA)

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function[32]. It is often used when the search space is discrete, as is the optimization problem encountered in hospital bed allocation. To clarify how SA can be applied to the problem of minimizing $E(s)$, we, we will first define the notation[32] used in the algorithm, as outlined in Table 3.5.

With the notation defined in Table 3.5, the acceptance probability function for SA can be calculated using the equation $P(e, e_{new}, T) = e^{-\Delta E/T}$. In this study, states with smaller energy are more desirable than those with greater energy because they indicate less total nonzero wait time. The SA algorithm for optimizing bed allocation in the hospital system can then be simplified as follows:

As noted in [33], the probability of SA converging to a global optimum increases with a sufficiently large initial T. However, finding the exact global optimum can be computationally expensive and requires a well-designed cooling schedule. In practice, we often settle for an approximate optimal solution, which is typically adequate.

Calculating the energy for current state and proposal neighbor site is essential for implementing SA. When optimizing bed allocation in a hospital with a static arrival rate, the energy can be determined using either queueing theory (M/M/k queue analysis with formula $E(W|W > 0) = \frac{1}{(1-\rho)k\mu}$) or simulation. However, for more realistic scenarios with

Algorithm 2 Simulated Annealing for Bed Allocation Optimization

Initialize parameters: Set the initial temperature T_0, cooling rate α, and stopping temperature T_{min}
Initialize state: Select an arbitrary initial bed allocation s_0 and compute the energy $E(s_0)$
Set the current state $s = s_0$ and temperature $T = T_0$
while $T > T_{min}$ **do**
    Generate new state: Randomly select one patient stream and increase its bed allocation by 1 or 2 beds
    Simultaneously, select another stream (excluding the previous) and decrease its bed allocation by 1 or 2 beds
    Calculate the energy of the new state $E(s_{\text{new}})$
    Acceptance criterion:
    if $\Delta E = E(s_{\text{new}}) - E(s) < 0$ **then**
        Accept the new state and update $s = s_{\text{new}}$
    else
        Accept s_{new} with probability $P(e, e_{\text{new}}, T) = e^{-\Delta E/T}$
        if accepted **then**
            Update current state $s = s_{\text{new}}$
        end if
    end if
    Update temperature: $T = T \cdot \alpha$
end while
Return the optimized bed allocation and minimum energy $E(s)$

dynamic arrival rates, simulation is typically employed to calculate the state energy, as queueing theory for M/M/k queue cannot accurately model such fluctuations.

With the mathematical model and SA algorithm for bed optimization in place, the next chapter will present a real-world case study to illustrate their application in a practical setting.

Nomenclature	Description	Application to the Proposed Problem
T_0	initial system temperature	initialize to a high value
s_0	initial state of system	the combined number of beds at each service department in initial state
s	current state of system	the combined number of beds at each service department in current state
s_{new}	neighboring state of system	the combined number of beds at each service department in neighboring state
e	$E(s)$, energy at current state	$\sum_{i=1}^{N}[E[W_i \mid W_i > 0]]^2$
e_{new}	$E(s_{new})$, energy at proposal neighbor site	$\sum_{i=1}^{N}[E[W_i^{new} \mid W_i^{new} > 0]]^2$
ΔE	energy change	$\sum_{i=1}^{N}[E[W_i^{new} \mid W_i^{new} > 0]]^2$ - $\sum_{i=1}^{N}[E[W_i \mid W_i > 0]]^2$
T	current system temperature	update at each iteration

Table 3.5: Simulated Annealing (SA) Parameters

CHAPTER 4

IMPLEMENTATION AND CASE STUDY

4.1 Introduction

Drawing upon the queueing theory and simulation modeling outlined in Chapter 3, we constructed hospital operations models grounded in a realistic scenario. We then juxtaposed the metrics derived from simulations with those from queueing theory, further formulating an optimization problem to optimize hospital bed allocation given a fixed total number of beds. To obtain optimal solutions, we employed the simulated annealing algorithm (SA). In the subsequent section, we delve into the specifics of a hospital case study.

4.2 Case Study Description

This section presents an in-depth portrayal of a case study within a hospital setting[1]: This case study examines a hypothetical hospital with a fixed number of beds and eight specialized service departments (also referred to as service centers) for general patients. Each department has a unique bed setup, catering to a distinct medical specialty, such as Surgery, Cardiology, General Medicine, Respiratory, Orthopedic, Gastroenterology-Endocrine, Oncology, and Renal Disease. At each service department, patients arrive and request beds [2]. We assume that patient arrivals at each service department follow a non-homogeneous Poisson process (NHPP) with a mean arrival rate that varies constantly

[1]This investigation leverages a simplified model of a hospital inpatient operations, where readmission of patients is excluded. Although this simplified model does not fully capture the real-world complexities, it serves as a robust foundation for mathematical modeling, and a more elaborate representation of hospital operations can be similarly formulated and analyzed based on this framework.

[2]For brevity, we refer to this as patients arriving at each service department.

throughout the day, exhibiting a 24-hour periodicity. Upon arrival at each service department, each patient is either assigned directly to a bed or must wait until a bed becomes available. Once assigned a bed, the patient remains in the department for service. The service time at each service department is assumed to be exponential distributed. After treatment, patients are discharged. All departments form a network within the hospital's fixed bed capacity, and patient flow in each department is regulated by arrival rates and service rates, as well as the number of available beds.

Terminology: Throughout this paper, the terms 'customer', 'arrival', and 'patient' are used interchangeably. Similarly, 'bed' and 'server' are synonymous, as are 'system time' and 'sojourn time'. Additionally, 'service department' and 'patient stream' are considered equivalent.

Service Department (Patient stream 1 to 8):

1. Patients arrive at each service department following a Poisson process with a dynamic arrival rate, which varies across medical specialties within the hospital. These rates exhibit fluctuations around their respective base values[3].
2. Each service department is equipped with a designated number of beds, and the service time for each patient follows an exponential distribution.
3. Throughout the metrics analysis, the total number of beds remains constant, ensuring a fixed capacity across all service departments.
4. Upon arrival at their designated service department, patients are either immediately admitted to an available bed for service or placed in a wait queue until a bed becomes vacant, at which point they are admitted and served.

[3]Specifically, the arrival rate of each medical specialty was modeled as a nonhomogeneous Poisson process, where the arrival rate = base arrival rate + 0.1 $\times$ base arrival rate $\times \sin\left(\frac{\pi t}{12}\right)$ in this investigation. This results in a 24-hour periodicity and a range between 0.9 and 1.1 times the base arrival rate.

Table 4.1 provides the base arrival rates[4], initial (non-optimized) bed allocations, and service rates pertaining to each stream within the hospital. We modeled the hospital's operations across all service departments using both queueing theory and simulation methodologies. Queueing theory provided the initial framework, with the base arrival rates serving as the static ones for analysis.

Service Department	**Base Arrival Rate λ (persons per hour)**	**Number of Beds k**	**Service Rate μ (person per hour)**
Stream 1	$\lambda_1 = 24.5/24 \approx 1.021$	$k_1 = 65$	$\mu_1 = 1/(2.37 * 24) \approx 0.0176$
Stream 2	$\lambda_2 = 37/24 \approx 1.542$	$k_2 = 119$	$\mu_2 = 1/(3.02 * 24) \approx 0.0138$
Stream 3	$\lambda_3 = 26/24 \approx 1.083$	$k_3 = 115$	$\mu_3 = 1/(4.09 * 24) \approx 0.0102$
Stream 4	$\lambda_4 = 19/24 \approx 0.792$	$k_4 = 62$	$\mu_4 = 1/(2.89 * 24) \approx 0.0144$
Stream 5	$\lambda_5 = 16.5/24 \approx 0.688$	$k_5 = 62$	$\mu_5 = 1/(3.27 * 24) \approx 0.0127$
Stream 6	$\lambda_6 = 14.1/24 \approx 0.588$	$k_6 = 57$	$\mu_6 = 1/(3.51 * 24) \approx 0.0119$
Stream 7	$\lambda_7 = 9.8/24 \approx 0.408$	$k_7 = 65$	$\mu_7 = 1/(5.56 * 24) \approx 0.0075$
Stream 8	$\lambda_8 = 16.3/24 \approx 0.679$	$k_8 = 85$	$\mu_8 = 1/(4.63 * 24) \approx 0.009$

Table 4.1: The Parameters List

4.3 Modeling the Hospital with Queueing Theory

To embark on our analysis of the hospital, we initially put to use M/M/k queueing analysis. This approach facilitated an approximation of the hospital's dynamics, incorporating

[4]Given that this work does not focus on raw data collection and preliminary analysis, we have adopted the hospital scenario described in paper[33] as an example, directly utilizing its summarized arrival rate parameters for illustrative purposes.

varying arrival rates that were estimated from base arrival rates. Alongside the specific bed allocation and service rates across the patient streams, we have compiled all pertinent parameters in Table 4.1. Utilizing the formulas inherent to the M/M/k queueing model, we calculated pivotal steady-state performance metrics, including the offered load, utilization, mean wait time for blocked arrivals (aliased as mean nonzero wait time), the probability of zero patients in the system, Erlang C value (signifying the probability of an arrival finding all servers busy), mean wait time, mean queue length, mean system length, mean service time, mean sojourn time, and throughput (an indicator that reflects the mean rate at which patients are served and exit the system), for each individual service department.

4.3.1 Formulas for M/M/k Queueing Analysis

In line with queueing theory conventions, we adopt the following notation:

1. a_1, ..., a_8 signify the offered load for streams 1 through 8, respectively
2. ρ_1, ..., ρ_8 represent the utilization of streams 1 through 8, respectively
3. $E(W_1|W_1 > 0)$, ..., $E(W_8|W_8 > 0)$ denote the mean wait time for blocked patients in streams 1 to 8, respectively
4. π_{01}, ..., π_{08} indicate the probability of zero patients in streams 1 through 8, respectively
5. P_{Q1}, ..., P_{Q8} are the Erlang C values for streams 1 to 8, reflecting the probability of an arrival finding all servers busy in each stream
6. $E(W_1)$, ..., $E(W_8)$ represent the mean wait times for streams 1 to 8, respectively
7. L_{q1}, ..., L_{q8} denote the mean queue lengths for streams 1 through 8, respectively
8. L_1, ..., L_8 signify the mean system lengths for streams 1 to 8, respectively, and

9. τ_1, ..., τ_8 represent the mean service times for streams 1 through 8, respectively
10. S_1, ..., S_8 denote the mean system times for streams 1 to 8, respectively
11. T_1, ..., T_8 indicate the mean throughputs for streams 1 through 8, reflecting the average rate at which patients are served and leave the system in each stream

When the M/M/k system attains a stable state (i.e. $\rho < 1$), we derived the desired performance metrics for this hospital by applying the M/M/k queueing model.

1. Mean service time (τ):
$$\tau = \frac{1}{\mu} \tag{4.1}$$
2. Offered load to the system (a):
$$a = \lambda\tau \tag{4.2}$$
3. Utilization in the system (ρ):
$$\rho = \frac{a}{k} \tag{4.3}$$
4. The probability of zero patients (π_0) in the system:
$$\pi_0 = \left[\sum_{i=0}^{k-1} \frac{(k\rho)^i}{i!} + \frac{(k\rho)^k}{k!(1-\rho)}\right]^{-1} \tag{4.4}$$
5. the probability of i patients in the system (π_i) is calculated by the formula:
$$\pi_i = \begin{cases} a^i \frac{1}{i!}\pi_0 & \text{if } i < k \\ a^i \frac{1}{k!}\left(\frac{1}{k}\right)^{i-k}\pi_0 & \text{if } i \geq k \end{cases} \tag{4.5}$$
6. Erlang-C Formula for Probability of wait (P_Q) [5]:
$$P_Q = \frac{(k\rho)^k}{k!(1-\rho)}\pi_0 \tag{4.6}$$

[5]In real case for every integer $k > a$, we calculate the value of P_Q in simulation by the following way: $P_Q = \frac{kB(k,a)}{k-a[1-B(k,a)]}$, where $B(k,a) = \frac{aB(k-1,a)}{k+aB(k-1,a)}$, $B(0,a) = 1$, a is the offered load[34].

7. The wait time probability ($P[W > t]$)[6]:

$$P[W > t] = P_Q * e^{-(1-\rho)k\mu t} \tag{4.7}$$

8. The nonzero wait time probability ($P[W > t|W > 0]$):

$$P[W > t|W > 0] = e^{-(1-\rho)k\mu t} \tag{4.8}$$

9. Mean wait Time in the queue ($E(W)$):

$$E(W) = \frac{P_Q}{(1-\rho)k\mu} \tag{4.9}$$

10. Mean nonzero wait time ($E(W|W > 0)$):

$$E(W|W > 0) = \frac{1}{(1-\rho)k\mu} \tag{4.10}$$

11. Mean number of customers in the queue (L_q):

$$L_q = \lambda E(W) \tag{4.11}$$

12. Mean number of customers in the system (L):

$$L = L_q + \frac{\lambda}{\mu} \tag{4.12}$$

13. Mean system time (S):

$$S = \frac{L}{\lambda} = E(W) + s \tag{4.13}$$

14. Throughput of the system (T):

$$T = \lambda \tag{4.14}$$

For the eight distinct patient streams, we computed the metric values by integrating the corresponding formulas with base arrival rates, the number of servers, and the service rates. The outcomes of these computations were displayed in Table 4.2. To facilitate

service department	a (Offered Load)	ρ (Utilization)	$E(W\|W>0)$ (Mean wait by those who are blocked)	π_0 (Probability of zero patients)	P_Q (An arrival finds all servers busy)	$E(W)$ (Mean wait time)	L_q (Mean queue length)	L (Mean system length)	τ (Mean service time)	S (Mean system time)	T (Throughput)
Stream 1	$a_1 = \lambda_1/\mu_1 \approx 58.065$	$\rho_1 = a_1/k_1 \approx 0.893$	$E(W_1\|W_1>0) = \frac{1}{(1-\rho_1)k_1\mu_1} \approx 8.202$	$\pi_{01} = \left[\sum_{i=0}^{k_1-1} \frac{(k_1\rho_1)^i}{i!} + \frac{(k_1\rho_1)^{k_1}}{k_1!(1-\rho_1)}\right]^{-1} \approx 5.445 \times 10^{-26}$	$P_{Q1} = \frac{(k_1\rho_1)^{k_1}}{k_1!(1-\rho_1)}\pi_{01} \approx 0.279$	$E(W_1) = \frac{P_{Q1}}{(1-\rho_1)k_1\mu_1} \approx 2.290$	$L_{q1} = \lambda_1 E(W_1) \approx 2.338$	$L_1 = L_{q1} + a_1 \approx 60.403$	$\tau_1 = \frac{1}{\mu_1} \approx 56.880$	$S_1 = E(W_1) + \tau_1 \approx 59.170$	$T_1 = \lambda_1 \approx 1.021$
Stream 2	$a_2 \approx 111.740$	$\rho_2 \approx 0.939$	$E(W_2\|W_2>0) \approx 9.983$	$\pi_{02} \approx 2.435 \times 10^{-49}$	$P_{Q2} \approx 0.391$	$E(W_2) \approx 3.900$	$L_{q2} \approx 6.012$	$L_2 \approx 117.752$	$\tau_2 = \frac{1}{\mu_2} \approx 72.480$	$S_2 \approx 76.380$	$T_2 \approx 1.542$
Stream 3	$a_3 \approx 106.340$	$\rho_3 \approx 0.925$	$E(W_3\|W_3>0) \approx 11.335$	$\pi_{03} \approx 5.770 \times 10^{-47}$	$P_{Q3} \approx 0.308$	$E(W_3) \approx 3.489$	$L_{q3} \approx 3.780$	$L_3 \approx 110.120$	$\tau_3 = \frac{1}{\mu_3} \approx 98.160$	$S_3 \approx 101.649$	$T_3 \approx 1.083$
Stream 4	$a_4 \approx 54.910$	$\rho_4 \approx 0.886$	$E(W_4\|W_4>0) \approx 9.783$	$\pi_{04} \approx 1.293 \times 10^{-24}$	$P_{Q4} \approx 0.259$	$E(W_4) \approx 2.537$	$L_{q4} \approx 2.009$	$L_4 \approx 56.919$	$\tau_4 = \frac{1}{\mu_4} \approx 69.360$	$S_4 \approx 71.897$	$T_4 \approx 0.792$
Stream 5	$a_5 \approx 53.955$	$\rho_5 \approx 0.870$	$E(W_5\|W_5>0) \approx 9.755$	$\pi_{05} \approx 3.461 \times 10^{-24}$	$P_{Q5} \approx 0.206$	$E(W_5) \approx 2.011$	$L_{q5} \approx 1.382$	$L_5 \approx 55.337$	$\tau_5 = \frac{1}{\mu_5} \approx 78.480$	$S_5 \approx 80.491$	$T_5 \approx 0.688$
Stream 6	$a_6 \approx 49.491$	$\rho_6 \approx 0.868$	$E(W_6\|W_6>0) \approx 11.219$	$\pi_{06} \approx 2.989 \times 10^{-22}$	$P_{Q6} \approx 0.217$	$E(W_6) \approx 2.432$	$L_{q6} \approx 1.429$	$L_6 \approx 50.920$	$\tau_6 = \frac{1}{\mu_6} \approx 84.240$	$S_6 \approx 86.672$	$T_6 \approx 0.588$
Stream 7	$a_7 \approx 54.488$	$\rho_7 \approx 0.838$	$E(W_7\|W_7>0) \approx 12.694$	$\pi_{07} \approx 2.111 \times 10^{-24}$	$P_{Q7} \approx 0.115$	$E(W_7) \approx 1.454$	$L_{q7} \approx 0.594$	$L_7 \approx 55.082$	$\tau_7 = \frac{1}{\mu_7} \approx 133.440$	$S_7 \approx 134.894$	$T_7 \approx 0.408$
Stream 8	$a_8 \approx 75.469$	$\rho_8 \approx 0.888$	$E(W_8\|W_8>0) \approx 11.659$	$\pi_{08} \approx 1.571 \times 10^{-33}$	$P_{Q8} \approx 0.203$	$E(W_8) \approx 2.364$	$L_{q8} \approx 1.605$	$L_8 \approx 77.074$	$\tau_8 = \frac{1}{\mu_8} \approx 111.120$	$S_8 \approx 113.484$	$T_8 \approx 0.679$

Table 4.2: The Theoretical Statistics by Queueing Theory

reproducibility, Appendix A contains the Python script that generates these theoretical predictions based on the M/M/k queueing model framework.

Having established these theoretical predictions, we progressed in the next section to develop a simulation model tailored specifically for the hospital with dynamic arrival rates. This model generated simulation outcomes, which were then meticulously analyzed and compared against the the theoretical predictions outlined in Table 4.2.

4.4 Simulation Modeling and Comparative Analysis with Queueing Theory

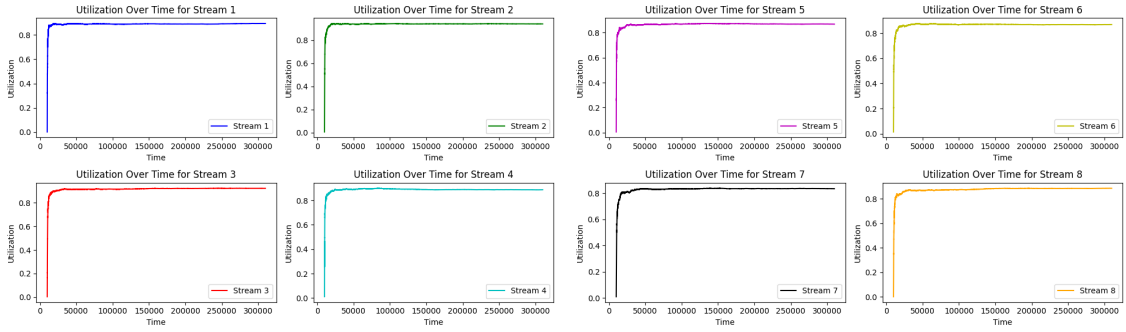
In this section, we introduced a simulation model that mirrors the hospital's operations with dynamic arrival rates, augmenting the theoretical insights from queueing theory

[6]Refer to [34] for 4.7, 4.8, 4.9, and 4.10.

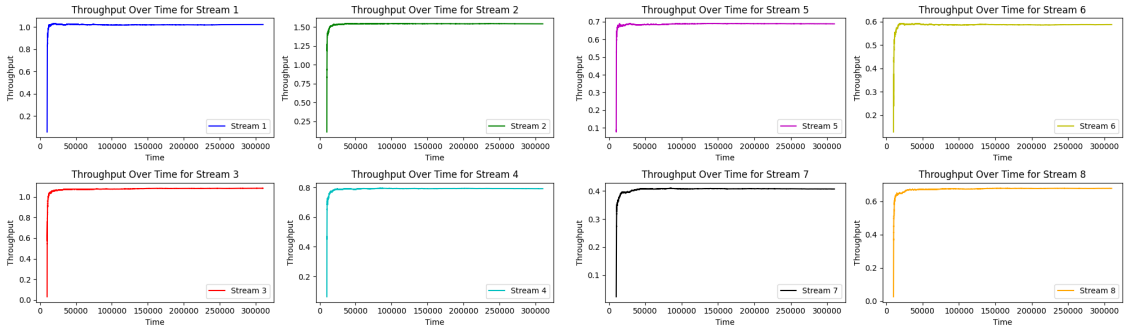
modeling. Our simulation analysis concentrated on two categories of metrics: those pertaining to statistical comparison and those geared towards time-series analysis. The former included metrics such as nonzero wait times, wait times, service times, and sojourn times (wait time in queue plus service time), where rigorous statistical tests were employed to contrast the distributions from our simulation model (M(t)/M/k queues) against the theoretical distributions predicted by M/M/k queueing theory. The latter category delved into the dynamic performance, examining queue lengths, system lengths, throughputs, and utilizations. To fortify the credibility of our findings, we calculated confidence intervals for both categories. Through a comparative analysis of simulation results and theoretical calculations in Table 4.2 by queueing theory, we discerned the merits and limitations of each modeling approach, thereby enhancing our comprehension of the hospital's operations and guiding future modeling endeavors.

4.4.1 Determining the Warmup Period for Simulation Accuracy

To mitigate the disruptive impact of initial transient effects and guarantee the precision of our simulation results, we at first created an extensive simulation exercise to identify an appropriate warmup duration. The intricate Python script for this purpose is appended in Appendix B for reference. Through meticulous analysis of utilization and throughput graphs across the eight streams (refer to Figure 4.1), we conclusively determined that a 10,000-hour warmup period was imperative. Following this determination, data collection was initiated, seamlessly spanning the entirety of the simulation's duration. These data were then archived for detailed metric analysis at a later state (the script for this analysis is provided in Appendix C). Our initial analytical investigation began with the service times and nonzero wait times observed within the M(t)/M/k queues.



Utilization: Stream 1-4 Utilization: Stream 5-8



Throughput: Stream 1-4 Throughput: Stream 5-8

Figure 4.1: Utilization and Throughput Performances Over Time

4.4.2 Metrics Analysis: Service Times and Nonzero wait Times

In an M/M/k queue, queueing theory posits that service time τ adheres to an exponential distribution parameterized by service rate μ, exhibiting the memoryless property. Additionally, queueing theory furnishes the formula for the probability $P[W > t|W > 0]$ (4.8) that a patient will experience a wait time. This formula highlights that the nonzero wait time also conforms to an exponential distribution. While queueing theory offers analytical methodologies for M/M/k queues in steady-state, these methodologies are confined to static arrival rates. When the arrival rate becomes dynamic, akin to an M(t)/M/k queue, the analyses may lose direct applicability or may yield only approximate results owing to increased complexity. Given queueing theory's prediction that both service time and the nonzero wait time in an M/M/k queue adhere to exponential distributions, we postulated that their distributions from simulation for the M(t)/M/k queue would exhibit

analogous statistical characteristics.

Initially, we concentrated on service times, presuming that the analysis patterns for nonzero wait times would mirror those of service times. However, the simulations revealed a slight divergence in the nonzero wait time distribution for the M(t)/M/K queue compared to the theoretical nonzero wait time distribution for M/M/K queue. Therefore, we started a simulation analysis for both service times and nonzero wait times to gain a more nuanced understanding of these metrics in the context of a hospital with dynamic arrivals. We began by examining the service time of the simulation.

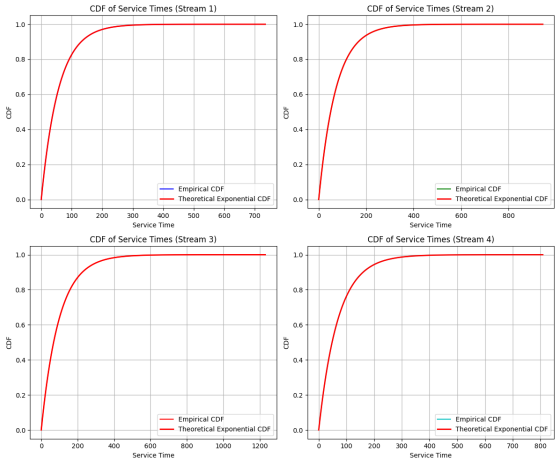
Service Time

To evaluate the distribution of service times, we gathered a dataset from the simulation and constructed empirical cumulative distribution functions (CDFs). Following this, we fitted these empirical CDFs to the corresponding theoretical distributions and visualized the results in Figure 4.2. We then interpreted the depicted CDFs and distributions in relation to the service times. (The comprehensive Python script for this analysis is appended in Appendix D.)

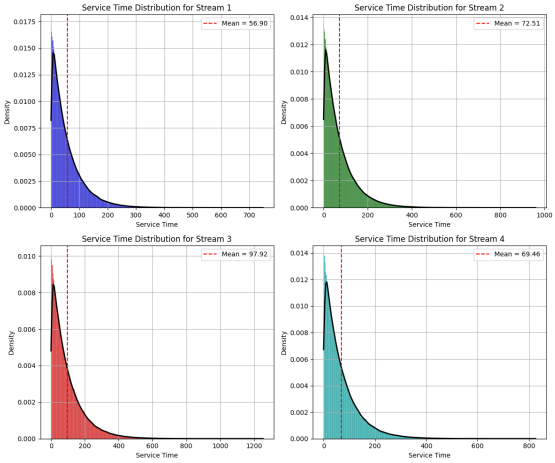
Interpreting Service Time CDFs and Distributions

1. As per the plots, the service time distributions for the different streams bear a resemblance to exponential distributions as expected.
2. The empirical CDFs and the theoretical CDFs are perfectly overlapping, which again implies that the exponential distribution is a suitable fit for the service times data.

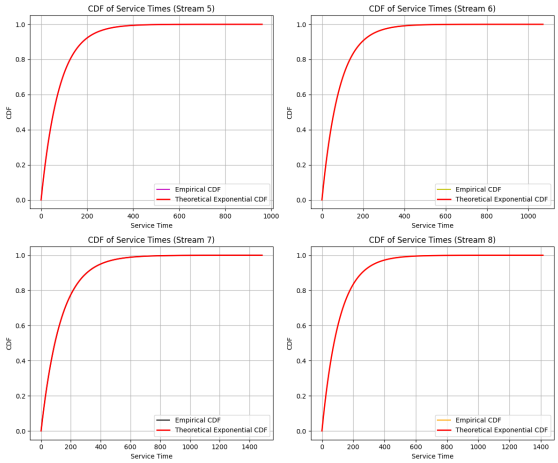
Despite the visual similarity implying that the service time at each service department follows an exponential distribution, rigorous statistical analysis is necessary to validate



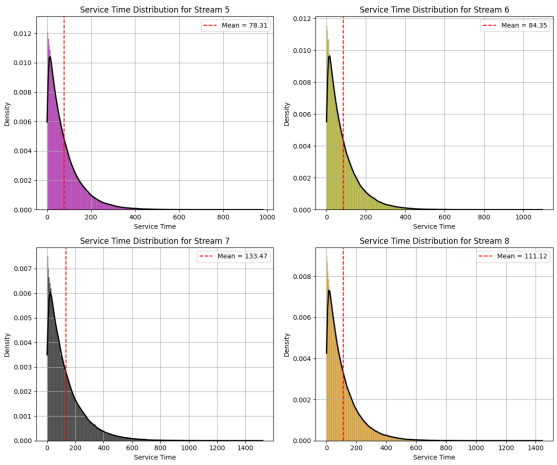
Comparing CDFs of Service Time: Stream 1-4



Service Time Distributions: Stream 1-4



Comparing CDFs of Service Time: Stream 5-8



Service Time Distributions: Stream 5-8

Figure 4.2: CDFs and Distributions for Service Times

this hypothesis. To that end, we performed a goodness-of-fit test for each stream to ascertain whether or not the service time adheres to an exponential distribution. For this purpose, we employed statistical tests like the Kolmogorov-Smirnov (K-S) test and computed 95% confidence intervals for the mean service times based on simulations. Using these results, we performed an analysis to compare the queueing theory predictions with simulation outcomes.

Service Department	95% CI (hours)[7]	Mean Values (hours)[8] (by Simulation, with Dynamic Arrival Rates)	Mean Values (hours)[9] (by Queueing Theory, with Static Arrival Rates)	Scales[10]	p-values[11]
Stream 1	(56.674, 56.872)	≈ 56.774	≈ 56.880	≈ 56.897	0.941
Stream 2	(72.085, 72.306)	≈ 72.195	≈ 72.480	≈ 72.511	0.946
Stream 3	(97.336, 97.703)	≈ 97.519	≈ 98.160	≈ 97.921	0.764
Stream 4	(68.916, 69.210)	≈ 69.063	≈ 69.360	≈ 69.457	0.160
Stream 5	(77.673, 78.133)	≈ 77.903	≈ 78.480	≈ 78.306	0.484
Stream 6	(83.655, 84.050)	≈ 83.804	≈ 83.852	≈ 84.353	0.996
Stream 7	(132.055, 132.980)	≈ 132.518	≈ 133.440	≈ 133.466	0.667
Stream 8	(110.336, 110.865)	≈ 110.601	≈ 111.120	≈ 111.125	0.385

Table 4.3: The Simulation Data Analysis for Service Times

Analysis of Theoretical Computations and Empirical Findings/Simulation Results

Theoretical Computations:

Queueing theory indicates that in the M/M/k queue, service time τ follows an exponential distribution characterized by the service rate μ. Utilizing the given service rate μ from Table 4.1, we computed the mean service times for each stream within the hospital. These mean service times were then displayed in a dedicated column of Table 4.3.

Observation of Simulation Results:

Table 4.3 further included the 95% confidence intervals and mean service times derived from the simulation involving dynamic arrival rates. In addition, the K-S test re-

[7]Refer to Appendix E for Python script
[8]Refer to Appendix E for Python script
[9]Refer to Table 4.2
[10]Refer to Appendix D for Python script
[11]Refer to Appendix D for Python script

sults for the eight streams simulated with dynamic arrival rates were presented in the same table. Our analysis of these results proceeded as follows:

1. All streams show simulated mean values that are slightly smaller than the theoretical values. This suggests a consistent pattern where the dynamic arrival rates slightly reduce the average service times, but the differences are minimal.
2. The theoretical mean values fall within or close to their respective 95% confidence intervals (CIs), showing that the theoretical model's assumptions about service times (based on static arrival rates) reasonably align well with the empirical data from the simulation.
3. The CIs are quite narrow across all streams, suggesting a high level of precision in the simulation results and relatively low variability in the service times.
4. The scale values are very close to the theoretical mean values across all streams, supporting the hypothesis that the service times may follow an exponential distribution.
5. All streams have relatively high p-values, and all of the p-values from the K-S test are larger than 0.05, suggesting that an exponential distribution is a reasonable fit for the service times in these streams.

Conclusion:

The simulation results for M(t)/M/k queues closely match the theoretical predictions for service times in M/M/k queues. This suggests that an exponential distribution accurately models the service times in the M(t)/M/k systems.

Overall, this outcome implies that the service processes in the M(t)/M/k queue are likely memoryless, characterized by constant service rates, thereby simplifying perfor-

mance prediction and management. This suggests that the system's service time metrics remain stable under the dynamic conditions in this investigation.

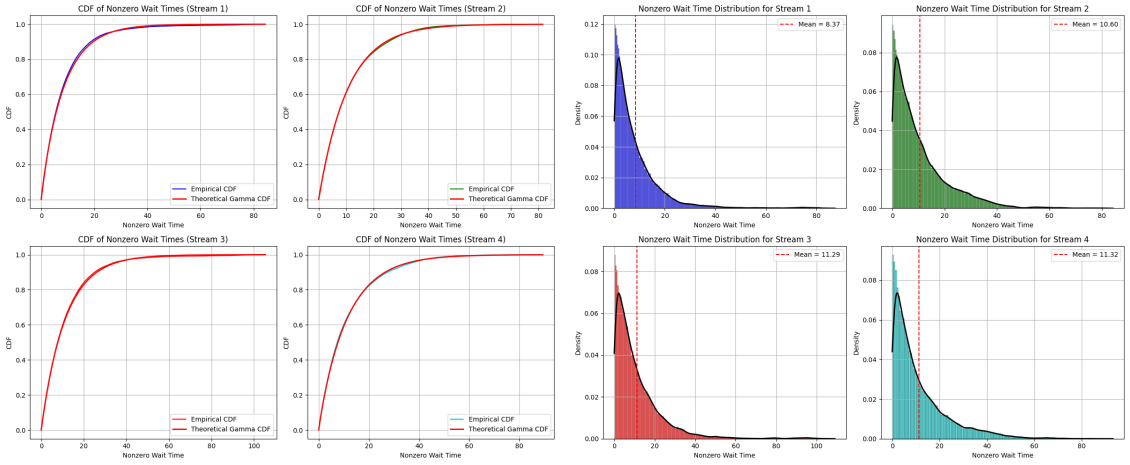
Next, we shifted our focus to analyzing nonzero wait times.

Nonzero wait Time

We collected data on nonzero wait times and compared empirical CDFs to theoretical CDFs, utilizing the gamma distribution due to its general characteristics and greater flexibility. Figure 4.3 illustrated the fitted results of CDFs and distributions. The detailed Python script for this analysis was provided in Appendix F. Additionally, we interpreted the displayed CDFs and distributions in the context of nonzero wait times.

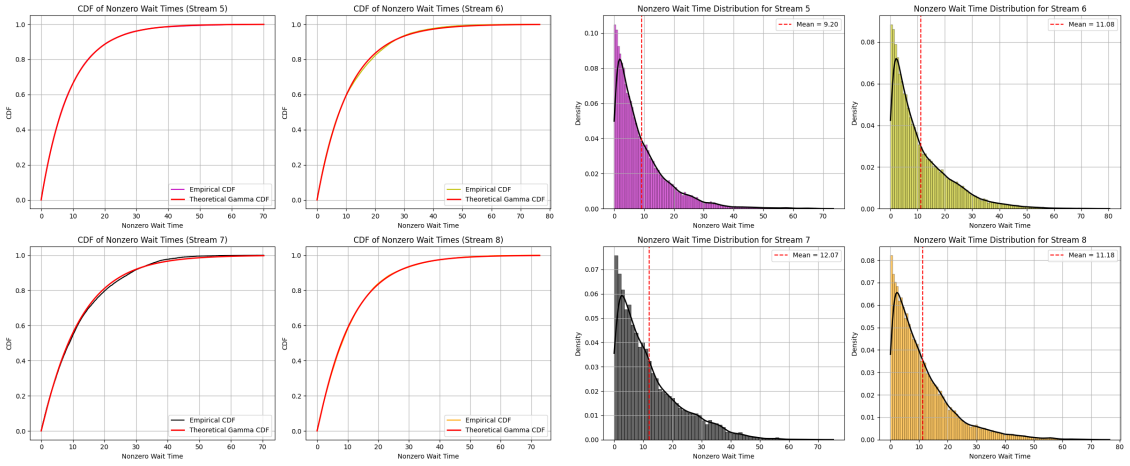
Interpretation of Nonzero wait Times: CDFs and Distributions

1. As per the plots, the nonzero wait time distributions for the different streams bear a resemblance to exponential distributions.
2. In some streams, like Stream 7 and Stream 2, there might be very slight deviations between the empirical and theoretical CDFs at the lower end of the nonzero wait time. However, the deviations are minor.
3. We observe that most of the the empirical CDFs and the theoretical CDFs are almost perfectly overlapping. This indicates that the nonzero wait times for these streams follow a Gamma distribution quite closely.
4. The shape of the CDF curves signifies the distribution of nonzero wait times. A steeper curve indicates that nonzero wait times are more concentrated toward lower values. For instance, in Stream 1, the CDF rises very quickly, suggesting that most nonzero wait times are relatively short. This is confirmed by the mean nonzero wait time from the nonzero wait time distribution. Conversely, Stream 7 has a more



(a) Comparing CDFs of Nonzero wait Time: Stream 1-4

(b) Nonzero wait Time Distributions: Stream 1-4



(c) Comparing CDFs of Nonzero wait Time: Stream 5-8

(d) Nonzero wait Time Distributions: Stream 5-8

Figure 4.3: CDFs and Distributions for Nonzero wait Time

gradual rise, indicating a wider spread in wait times, which is also confirmed by the mean nonzero wait time from the nonzero wait time distribution. These findings can inform our optimization efforts.

5. The fact that the CDFs line overlap well implies that the gamma distribution is a good fit for the nonzero wait times data.

Although these visual similarities suggest that the nonzero wait time may follow a gamma distribution, more specifically resembling an exponential distribution, rigorous

statistical analysis is still required to confirm this hypothesis. Given our goal of ascertaining whether the nonzero wait times in the M(t)/M/k queue follow an exponential distribution, as predicted by M/M/k queue theory, we conducted additional K-S tests to verify if the gamma distribution with a shape parameter of 1 holds for the nonzero wait times. With these results, we conducted an analysis to compare the predictions of queueing theory with the simulation results.

Analysis of the Theoretical Computations and the Empirical Findings/Simulation Results

Theoretical Computations:

Upon analyzing the nonzero wait time probability $P[W > t|W > 0]$ (4.8), we discovered that the nonzero wait time in an M/M/k queue conforms to an exponential distribution. Meanwhile, the mean nonzero wait times for each stream can be determined, as shown in Table 4.4.

Observation of Simulation Results:

By conducting the K-S test (specially for the gamma distribution) and confidence interval analysis on the collected simulation dataset, we incorporated these statistical findings into Table 4.4. Our analysis is as follows:

1. The 95% confidence intervals (CIs) for the dynamic arrival rates capture the range in which the mean nonzero wait time for M(t)/M/k queue is expected to lie with 95% confidence. The theoretical value (based on static arrival rates) serves as a bench-

[12]Refer to Appendix E for Python script
[13]Refer to Appendix E for Python script
[14]Refer to Table 4.2
[15]Refer to Appendix F for Python script
[16]Refer to Appendix F for Python script
[17]Refer to Appendix F for Python script

Service department	95% CI (hours)[12] (Simulation, Dynamic Arrival Rate)	Mean Values (hours)[13](Simulation, Dynamic Arrival Rate)	Mean Values (hours)[14] (by Queueing Theory, with Static Arrival Rates)	Scales[15]	Shapes[16]	p-values[17]
Stream 1	(7.772, 9.090)	≈ 8.431	≈ 8.202	≈ 8.885	≈ 0.942	$\approx 1.062 \times 10^{-19}$
Stream 2	(9.013, 10.482)	≈ 9.748	≈ 9.983	≈ 10.643	≈ 0.996	$\approx 1.037 \times 10^{-13}$
Stream 3	(9.858, 11.795)	≈ 10.827	≈ 11.335	≈ 11.525	≈ 0.979	$\approx 2.226 \times 10^{-13}$
Stream 4	(8.462, 9.571)	≈ 9.017	≈ 9.783	≈ 12.041	≈ 0.941	$\approx 8.701 \times 10^{-10}$
Stream 5	(8.815, 10.240)	≈ 9.527	≈ 9.755	≈ 9.282	≈ 0.991	≈ 0.330
Stream 6	(10.593, 12.206)	≈ 11.400	≈ 11.219	≈ 11.116	≈ 0.992	$\approx 1.001 \times 10^{-9}$
Stream 7	(10.994, 12.890)	≈ 11.942	≈ 12.694	≈ 11.376	≈ 1.061	≈ 0.001
Stream 8	(10.263, 12.103)	≈ 11.183	≈ 11.659	≈ 10.522	≈ 1.062	≈ 0.0001

Table 4.4: The Simulation Data Analysis for Nonzero wait Times

mark. The intervals mostly encompass or are close to the theoretical expectations. Notable exceptions include Streams 3, 4, 7, and 8, where the simulation means fall slightly outside the theoretical ranges, implying a potential effect of dynamic rates on these streams. indicating that the theoretical mean nonzero wait time is reasonably close to the empirical mean wait times observed with dynamic arrival rates.

2. Confidence intervals are reasonably narrow, indicating consistent results across all simulation replications.

3. The K-S test results for scale parameters are all within the 95% confidence intervals. The scale parameter generally correlates with the mean nonzero wait time, with higher scale values indicating longer nonzero waits. Stream 4 has the highest scale parameter (12.041), corresponding to its relatively high mean nonzero wait time.

4. The shape parameter is close to 1 for all streams, which indicates that the nonzero wait times are nearly exponentially distributed.

5. The p-value from the K-S test indicates how well the empirical distribution of nonzero wait times fits the theoretical gamma distribution. All but stream 5's p-

values are less than 0.05, indicating that the gamma distribution may not fit perfectly for these streams, despite visually good fits in the CDF plots. This indicates more variability and uncertainty due to the fluctuating arrival rates.

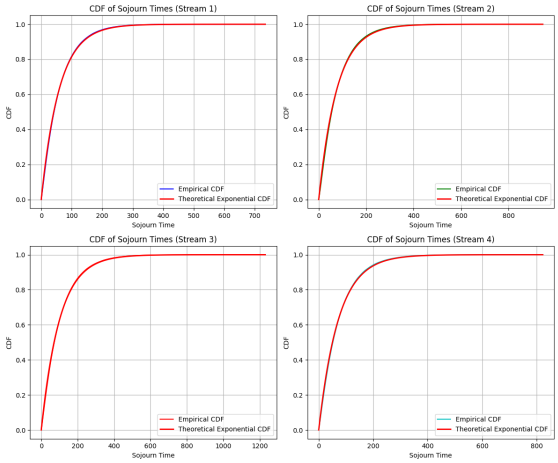
Conclusion:

The simulation aligns well with theoretical mean nonzero wait times, demonstrating that the dynamic arrival rates under the given pattern do not significantly disrupt expected mean values. However, the gamma distribution does not fit the nonzero wait times well for most streams, as indicated by the K-S test. This suggests that while the averages are stable, the distributional shape of the nonzero wait times varies, likely due to the impact of dynamic arrivals.

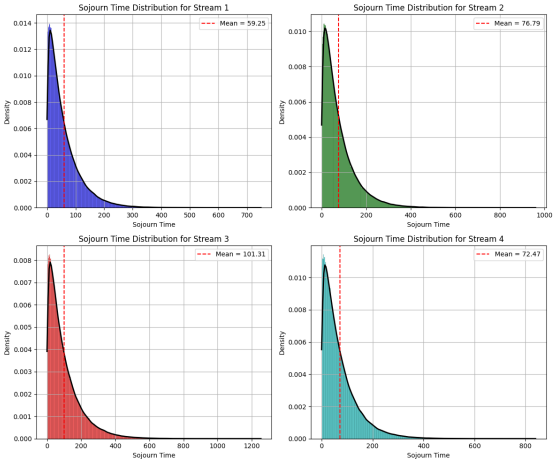
In general, the queueing model provides a rapid initial assessment of the nonzero wait times. In contrast, the simulation model offers a more realistic representation of the nonzero wait times across the patient streams. Future research may explore alternative distribution models or time-based effects to account for variability introduced by new dynamic arrival rate patterns.

4.4.3 Metrics Analyses: Sojourn Times and Wait Times

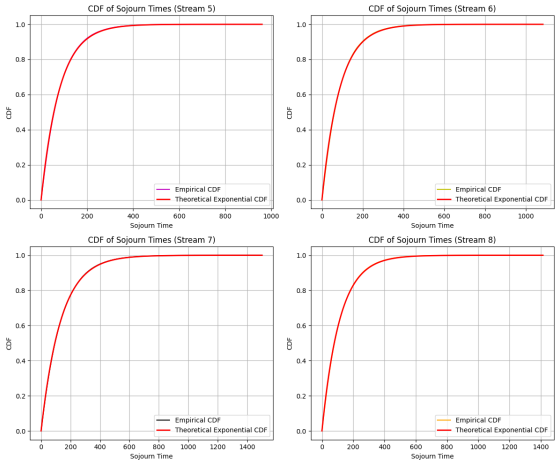
We subsequently analyzed both sojourn times and wait times, noting that both deviate from exponential distributions. In this investigation, we primarily focused on analyzing sojourn times, as the analysis for wait times follows a similar approach. As detailed in Appendix G, the Python script was utilized to analyze the CDFs and distributions of sojourn times, which are visualized in Figure 4.4. Below, we provided an interpretation of the CDFs and distributions in the figure.



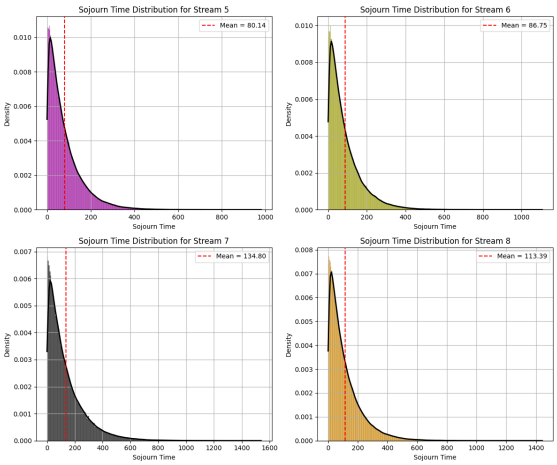
Comparing CDFs of Sojourn Time: Stream 1-4



Sojourn Time Distributions: Stream 1-4



Comparing CDFs of Sojourn Time: Stream 5-8



Sojourn Time Distribution: Stream 5-8

Figure 4.4: CDFs and Distributions for Sojourn Times

Interpreting Sojourn Time CDFs and Distribution

1. The plots suggest that the sojourn time distributions for the different streams closely resemble exponential distributions.
2. The empirical CDFs and the theoretical CDFs appear to overlap, indicating that the exponential distribution is a good fit for the sojourn times.

Although the results indicated a potential exponential distribution for the sojourn

times, further validation through K-S tests is essential to confirm this distribution conclusively. This is due to the fact that exponential distributions can often resemble other distributions with long tails, especially when visualized on a large scale. Moreover, we calculated the 95% confidence intervals for the mean sojourn time to assess how closely the simulation results align with the theoretical computations. With these results, we conducted a statistical analysis of the sojourn times and compared the results from queueing theory with those from the simulations.

Analysis of the Theoretical Computations and the Empirical Findings/Simulation Results

Theoretical Computations:

The wait time probability $P[W > t]$ (4.7) demonstrates that in M/M/k queues, even with constant arrival rates and exponential service times, the wait time distribution is influenced by queue dynamics and the number of servers. Consequently, the overall sojourn time distribution is typically more complex than a simple exponential distribution. We calculated the mean sojourn times for all patient streams using M/M/k queueing and presented these values in a column of Table 4.5.

Observation of Simulation Results:

Since the wait time is relatively short compared to the service time in this paper, the overall sojourn time distribution can appear similar to the exponential service time distribution. To verify this assumption, we conducted K-S tests and computed 95% confidence intervals for the mean sojourn times, creating Table 4.5. The K-S test results and confidence intervals in Table 4.5 were analyzed as follows:

[18]Refer to Appendix E for Python script
[19]Refer to Appendix E for Python script
[20]Refer to Table 4.2
[21]Refer to Appendix G for Python script
[22]Refer to Appendix G for Python script

Service department	95% CI (hours)[18] (Simulation, Dynamic Arrival Rate)	Mean Value (hours)[19] (Simulation, Dynamic Arrival Rate)	Mean Values (hours)[20] (by Queueing Theory, with Static Arrival Rates)	Scales[21]	p-values[22]
Stream 1	$(59.023, 59.766)$	≈ 59.395	≈ 59.170	≈ 59.246	$\approx 1.723 \times 10^{-114}$
Stream 2	$(75.872, 76.887)$	≈ 76.379	≈ 76.379	≈ 76.787	≈ 0
Stream 3	$(100.376, 101.499)$	≈ 100.938	≈ 98.160	≈ 101.308	$\approx 4.351 \times 10^{-107}$
Stream 4	$(71.128, 71.788)$	≈ 71.458	≈ 69.360	≈ 72.471	$\approx 7.015 \times 10^{-95}$
Stream 5	$(79.561, 80.387)$	≈ 79.974	≈ 78.480	≈ 80.140	$\approx 5.672 \times 10^{-34}$
Stream 6	$(86.193, 87.067)$	≈ 86.630	≈ 84.240	≈ 86.747	$\approx 1.329 \times 10^{-37}$
Stream 7	$(133.453, 134.648)$	≈ 134.051	≈ 133.440	≈ 134.801	≈ 0.0001
Stream 8	$(112.518, 113.497)$	≈ 113.007	≈ 111.120	≈ 113.385	$\approx 5.674 \times 10^{-27}$

Table 4.5: The Simulation Data Analysis for Sojourn Times

1. Most streams exhibit slightly higher simulated mean sojourn times than theoretical values, suggesting that dynamic arrival rates contribute to slightly longer sojourn times across most streams.
2. For most streams, the theoretical mean sojourn times based on static arrival rates fall within or near their respective 95% confidence intervals by simulation. This suggested that the simulated values are generally consistent with theoretical expectations under static conditions, though there is a slight upward deviation in several streams.
3. All streams have very low p-values, indicating that the K-S test rejects the hypothesis that the sojourn time at each service department follows an exponential distribution.
4. The scale values are close to the mean values for each stream, as expected for an exponential distribution fit.
5. The p-values are extremely low for all streams, indicating that the exponential distribution is not a good fit for the sojourn time data.

Conclusion:

The K-S test results indicate that the actual distribution of sojourn times deviates significantly from an exponential distribution. This finding suggests that the sojourn times exhibit more complex behavior than what an exponential model can capture.

After gaining insights into the sojourn times, we moved on to the queue lengths and the number of patients in the hospital.

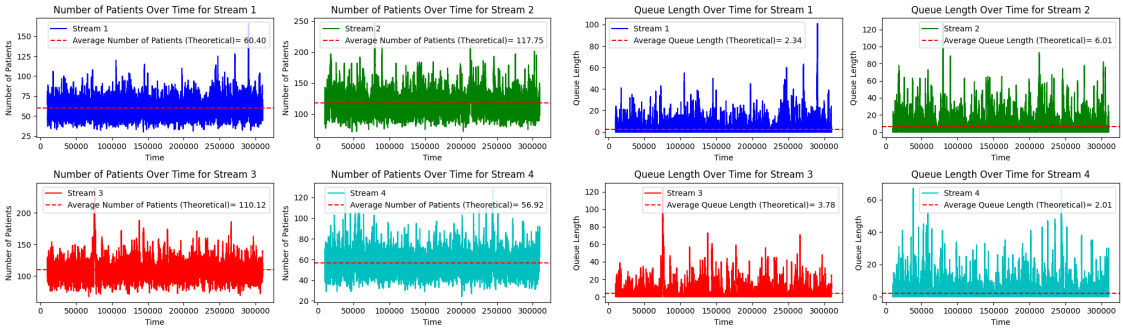
4.4.4 Metrics Analyses: Queue Lengths and the Number of Patients

In real-world scenarios, queue length and the number of patients (or system length) in are inherently dynamic. While queueing theory provides valuable insights into equilibrium conditions for M/M/k queues, the hospital's M(t)/M/k queues exhibit time-varying arrival rates, leading to transient congestion. Consequently, these factors result in fluctuations in queue and system lengths over time. Our analyses uncovered similar trends in both queue and system lengths, prompting us to concentrate solely on queue lengths for subsequent inquiries.

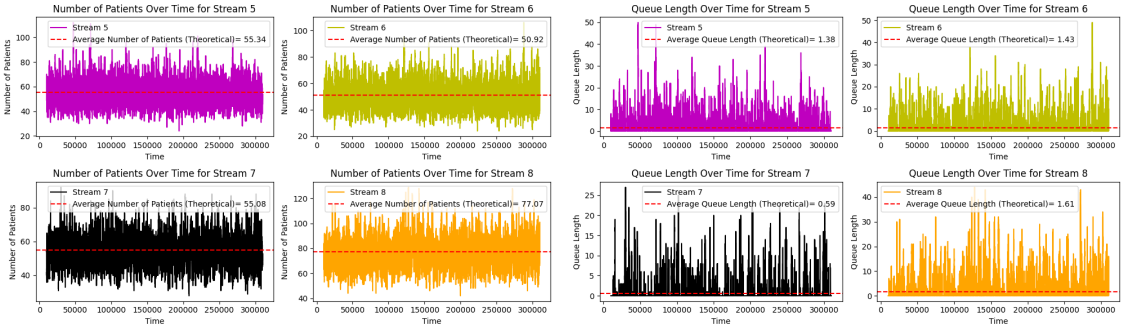
To understand the discrepancy between the mean queue length of an M(t)/M/k queue and the steady-state mean value of its corresponding M/M/k queue, we examined its behavior over time. Appendix C provides the detailed Python script for this analysis. Figure 4.5 plots variations in queue and system lengths over time across the eight patient streams, benchmarked against a baseline derived from a static arrival system. The subsequent analysis of queue lengths is as follows.

Interpreting the Plots of Queue Lengths over Time

1. All streams exhibit significant fluctuations in queue lengths over time. This vari-



System Lengths: Stream 1-4 Queue Lengths: Stream 1-4



System Lengths: Stream 5-8 Queue Lengths: Stream 5-8

Figure 4.5: System Lengths and Queue Lengths Over Time

ability is not only for queues with dynamic arrival rates, but also for queues with static arrival rates, where bursts of arrivals can temporarily increase queue lengths.

2. Each plot includes the mean queue length (marked by the red dashed line) from static arrival rates by queueing theory, which provides a summary of the typical queue size for each stream.

In summary, based on the plots of queue length above, we deducted that, despite the queue length in an M(t)/M/k fluctuating over time, it oscillates around a baseline, which corresponds to the value derived from the steady-state mean of an M/M/k queue.

Next, we compared the mean queue lengths obtained from queueing theory, simulation data, and the corresponding 95% confidence intervals. This analysis aimed to determine whether or not the simulation results accurately approximated the theoretical mean queue lengths, even under the influence of dynamic arrival rates. Table 4.6 shows these

statistical comparisons. Subsequently, we provide the following analysis of this table.

	L_{q1}	L_{q2}	L_{q3}	L_{q4}	L_{q5}	L_{q6}	L_{q7}	L_{q8}
Mean Value[23] (Static Arrival Rate)	2.338	6.012	3.780	2.009	1.382	1.429	0.594	1.605
Mean Value[24] (Dynamic Arrival Rate)	2.673	6.462	3.702	1.888	1.433	1.636	0.629	1.641
95% CI[25] (Dynamic Arrival Rate)	(2.351, 3.000)	(5.748, 7.176)	(3.188, 4.215)	(1.695, 2.081)	(1.247, 1.619)	(1.435, 1.837)	(0.539, 0.719)	(1.415, 1.867)

Table 4.6: Queue Length over Time

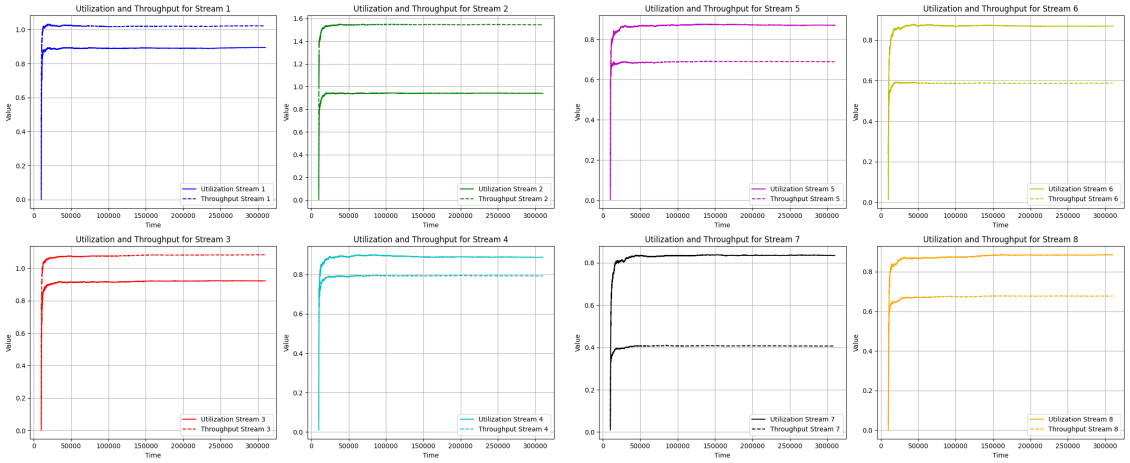
Analysis of the Theoretical Computations and Empirical Findings/Simulation Results

1. For all streams except stream 6, the theoretical mean values by queueing theory lie within their respective 95% confidence intervals, suggesting that the empirical queue lengths under dynamic arrival rates are generally consistent with the theoretical predictions.
2. Furthermore, most streams (five out of eight) have slightly higher mean queue lengths under dynamic conditions. This suggests that the variability introduced by dynamic arrival rates generally leads to increased queue lengths, though the effect is small in magnitude.

Conclusion:

In general, we observed that the queue lengths in M(t)/M/k queues stochastically fluctuated. Our analysis also demonstrated that the overall performance of queue lengths in M(t)/M/k queues was slightly higher than the theoretical mean values from static arrival rates, though the differences are small.

[23]See Appendix A for Python script
[24]See Appendix E for Python script
[25]See Appendix E for Python script



Utilizations and Throughputs: Stream 1-4 Utilizations and Throughputs: Stream 5-8

Figure 4.6: Utilizations and Throughputs Over Time

Lastly, we evaluated the utilization and throughput metrics within the hospital to gauge the overall system efficiency.

4.4.5 Metrics Analyses: Utilization and Throughput

The final two statistics, utilization and throughput for the queue, visually illustrate a queue's stability. To verify that these metrics follow a consistent pattern over time for each stream, we have plotted them together in Figure 4.6. Below is the analysis of these plots. Also, Appendix H contains the detailed Python script used for the plots.

Interpreting Utilizations and Throughputs Over Time

1. The utilization and throughput values stabilize over time, confirming that each patient stream attains a steady-state eventually.
2. Prior to reaching its steady-state, a patient stream in the hospital undergoes an initial transient phase known as the warmup period. We established a warmup period of 10,000 hours and colleted the simulation data only thereafter, thereby enhancing the

precision of the simulation results.

3. The utilization and throughput across different streams exhibit minimal differences.

According to M/M/k queueing theory, when the utilization is less than 1, the throughput is expected to match the arrival rate. Initially, we computed the mean values of utilization and throughput for each stream using queueing theory with static arrival rate. Additionally, we calculated the 95% confidence intervals for each stream's utilization and throughput. The following analysis discussed these values, which are presented in Table 4.7.

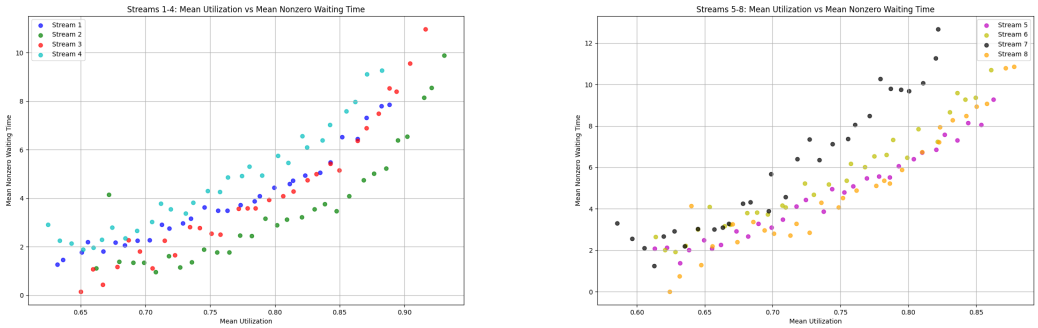
	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	ρ_8	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Mean Value[26] (Static Arrival Rate)	0.893	0.939	0.925	0.886	0.870	0.868	0.838	0.888	1.021	1.542	1.083	0.792	0.688	0.588	0.408	0.679
Mean Value[27] (Dynamic Arrival Rate)	0.888	0.932	0.913	0.876	0.860	0.863	0.829	0.878	1.000	1.509	1.053	0.774	0.672	0.576	0.395	0.659
95% CI[28] (Dynamic Arrival Rate)	(0.885, 0.890)	(0.930, 0.934)	(0.910, 0.915)	(0.873, 0.879)	(0.856, 0.863)	(0.860, 0.866)	(0.825, 0.833)	(0.876, 0.881)	(0.998, 1.003)	(1.506, 1.513)	(1.051, 1.055)	(0.771, 0.776)	(0.669, 0.674)	(0.574, 0.578)	(0.394, 0.397)	(0.657, 0.661)

Table 4.7: Utilization and Throughput

Analysis of the Theoretical Values and the Simulation Results

1. For all streams, the mean utilization under dynamic conditions is close to the static utilization values, indicating a reasonable alignment between the theoretical and empirical utilization. This also suggests that the system, when subject to varying arrival rates, has the same idle time as predicted by the static model.
2. The 95% confidence intervals of utilization are narrow for all streams, suggesting that the simulation results for utilization are stable and consistent.
3. The empirical throughput values under dynamic conditions follow the same pattern as utilization, which again confirmed the alignment between the theoretical and empirical throughput.

[26]See Appendix A for Python script
[27]See Appendix E for Python script
[28]See Appendix E for Python script



Stream 1-4 Stream 5-8

Figure 4.7: Scatter Plots of Utilization vs. Nonzero wait Time

Discussion:

The analysis shows a consistent and reasonable alignment between the static theoretical model and the dynamic simulation data for both utilization and throughput across all streams.

Queue theory establishes that for an M/M/k queue, the mean nonzero wait time, $E[W > t|W > 0]$ (4.10) is dependent on the utilization. Specifically, higher utilization leads to longer mean wait times. As utilization approaches 1, the mean wait time can be arbitrarily large.

Decreasing utilization is a viable strategy for reducing mean nonzero wait times. To examine this relationship empirically, we employed a Python script (Appendix I) for visual inspection and correlation analysis. The results are depicted in Figure 4.7 and Table 4.8.

	Stream 1	**Stream 2**	**Stream 3**	**Stream 4**	**Stream 5**	**Stream 6**	**Stream 7**	**Stream 8**
Correlation	0.967	0.864	0.955	0.961	0.982	0.971	0.958	0.950

Table 4.8: Correlation between Utilization and Nonzero wait Time

Analysis from Plots and Test Results

1. Across all streams, there is a clear positive relationship between mean utilization and mean nonzero wait time. This means that as utilization increases, nonzero wait time generally increases as well, and vice versa.
2. All streams exhibit a high positive correlation coefficient, which reinforces the observation from the scatter plots.
3. The positive trend in the scatter plots and the high correlation coefficients confirm that as server utilization increases, the system becomes more congested, leading to longer wait times. This is a fundamental characteristic of queueing systems.

General Conclusion

1. In all the metrics analyses, we found that the simulation results with time-varying arrival rates usually behave similarly to the ones with constant arrival rates.
2. Simulation results often have higher metrics values compared to the results from theoretical models, suggesting that the theoretical models may not fully account for the operational constraints present in real-world scenarios.
3. Because queueing theory does not account for fluctuations present in real-world systems, there is a discrepancy between theoretical values for M/M/k queue and simulation values for M(t)/M/k queue. We can ascertain that the simulation and theoretical models generally align. However, the metric values from the simulation models are often higher than those from queueing theory.
4. Theoretical models are valuable for initial estimates and understanding basic system behavior.

While theoretical and simulation models often diverge, real-world simulations are crucial for accurately understanding dynamic system behavior and optimizing performance. Nonetheless, as demonstrated above, employing queueing theory analysis with a static arrival rate can serve as a baseline for understanding and approximating optimal bed allocation solutions, especially when time-efficient insights are required.

4.5 Hospital System Optimization Analysis

A frequent inquiry for the hospital system is how to allocate a fixed number of beds optimally among different service departments to minimize the total mean nonzero wait time across the patient streams. This optimization analysis can assist the hospital in effectively distributing existing resources (beds), leading to improved performance and patient satisfaction.

4.5.1 Optimization Problem Development

Building upon the example presented in Section 4.2, we formulated the following nonlinear minimization problem. Using the parameters from Table 4.1 and the dynamic arrival rates described in Section 4.2, our goal is to minimize the total mean nonzero wait time across all the service departments.

- **Objective Function:**

  As discussed in Section 3.6.1, our goal is to ensure the same level across all service departments. Therefore, our objective is about the wait times in difference service departments. We aimed to minimize the total value of squared nonzero wait times

at each service department, which is represented mathematically as follows:

$$\min . \sum_{i=1}^{8} [E(W_i|W_i > 0)]^2 \tag{4.15}$$

Here, the number of the M/M/k service departments is 8, and $E(W_i|W_i > 0)$ is the mean nonzero wait time in the i^{th} M/M/k queue.

- **Constraints:**

1. *Mean Nonzero wait Time Constraints:*

$$E(W_i|W_i > 0) > 0, \quad i = 1, 2, ..., 8 \tag{4.16}$$

These inequalities ensure that all eight M/M/k queues' mean squared nonzero wait times must be greater than zero.

2. *Server constraints:*

$$\sum_{i=1}^{8} k_i = 630, \quad i = 1, 2, ..., 8 \tag{4.17}$$

$$0 \leq k_i \leq 630, \quad i = 1, 2, ..., 8, \quad \forall k_i \in integer \tag{4.18}$$

The equation maintains that the total number of servers across all M/M/k queues equal to the total number of servers in the entire M/M/k network. The inequalities ensure that the number of the server at each individual M/M/k queue satisfies its own constraints.

3. *Utilization Constraints:*

$$0 \leq \rho_i \leq 1, \quad i = 1, 2, ..., 8, \quad \forall k_i \in integer \tag{4.19}$$

Using the formula for calculating the utilization, these inequalities guarantee that the utilization falls within the range of 0 to 1.

4.5.2 Simulated Annealing (SA) Algorithm

In the context of hospital bed allocation with M(t)/M/k queues, we conducted a simulation to determine the state energy in SA. We observed that a high-quality initial solution is crucial for SA's success, as it can prevent premature convergence to local optima and significantly reduce search time.

Given the relationship between M/M/k and M(t)/M/k queues, we initially optimized a hospital system with static arrival rates using queueing theory. The optimal solution from this analysis served as the initial solution for the SA algorithm that we applied to the dynamic arrival rate problem. For the M/M/k queueing model, the utilization (ρ) and mean nonzero wait time ($E(W|W > 0)$) are given by $\rho = \frac{\lambda}{k\mu}$, $E(W|W > 0) = \frac{1}{(1-\rho)k\mu}$. Then, the optimization problem for the static arrival rate case can be formulated as:

$$\min . \sum_{i=1}^{8} [\frac{1}{(1-\rho_i)k_i\mu_i}]^2 \tag{4.20}$$

and 4.19 became

$$0 \leq \frac{1}{(1-\rho)k\mu} \leq 1 \tag{4.21}$$

The server constraints from Equations 4.17 and 4.18 remain unchanged.

Remark:

1. The bed allocation combination presented in Table 4.1 is merely an initial allocation example. Any combination that meets the condition $0 < \rho < 1$ for all streams is feasible. Using the formula $\rho = \frac{\lambda}{k\mu}$ and the data from Table 4.1, we calculated the range of possible beds for each stream and established an initial state for the static arrival rate problem.

In this study, we used the combination from Table 4.1 as the initial state and solved the

problem using the SA algorithm (See Python script of Appendix J). The optimum for these static arrival rates is shown in Table 4.9.

	Stream 1	**Stream 2**	**Stream 3**	**Stream 4**	**Stream 5**	**Stream 6**	**Stream 7**	**Stream 8**
Optimum	64	119	115	62	62	58	65	85

Table 4.9: Optimal Solution for M/M/k Queues

Starting with the optimal solution for the static arrival rates case, we applied the SA algorithm to address the nonlinear problem with dynamic arrival rates. For this scenario, the utilization (ρ) and mean nonzero wait time ($E(W|W > 0)$) were obtained through simulation rather than theoretical calculations. After running the simulation (See Python script of Appendix K), we found the optimum for the dynamic arrival rates case, as shown in Table 4.10. The objective decreased from 799 to 722, representing a 2.13% change.

	Stream 1	**Stream 2**	**Stream 3**	**Stream 4**	**Stream 5**	**Stream 6**	**Stream 7**	**Stream 8**
Optimum	68	118	113	62	62	59	64	84

Table 4.10: Optimal Solution for M(t)/M/k Queues

We observed that the optimal solution obtained from the SA algorithm differed between the static arrival rate and the dynamic arrival rate case. To be specific, the total nonzero wait times decreased from 799 hours to 772 hours, which is 2.13% change. This change was as anticipated and can be explained by the following:

1. The static arrival rates assume a constant rate of arrivals, which simplifies the problem but does not capture the variability and peaks in real-world scenarios. Dynamic arrival rates, however, account for fluctuations over time, providing a more realistic representation of the system.

2. Dynamic simulations captured complex interactions and dependencies between different variables which static models overlooked.

Therefore, we conclude that the solution from the SA algorithm for the dynamic arrival rate represents the optimal bed arrangement for the real-world hospital system. With the optimal bed allocation identified, we are now interested in doing further analysis on this optimal problem.

Sensitivity Analysis:

Assuming the hospital plans to increase its bed capacity by 10% (63 beds) in the upcoming procurement, we sought to optimize the allocation of these additional beds across various service departments. This does not imply an equal distribution across all service departments. Due to the varying impact of additional beds on each service department, optimizing the allocation of these new beds would require the application of SA algorithm. We employed SA algorithm to identify the optimal bed allocation strategy for a 10% increase in capacity. Our analysis revealed a significant reduction in the objective function, dropping from 905.94 hours to 234.82 hours under static arrival rates. This represents a decrease of approximately 74%. While it is important to note that this result is specific to the particular arrival rate pattern examined in this study, it nonetheless demonstrates the potential impact of a modest increase in bed capacity on patient wait times. This finding suggests that the hospital should carefully consider future bed procurement plans to enhance patient satisfaction.

Conclusion:

Queueing theory offers a preliminary approach to optimizing the hospital system under static arrival rates. This provides a foundation for addressing the more complex scenarios of dynamic arrival rates. By applying the SA algorithm to simulation results with dynamic arrival rates, we can identify optimal solution for real-world conditions.

CHAPTER 5

CONCLUSION

5.1 Conclusion

We formulated a mathematical and simulation model using a heuristic method tailored for a practical hospital scenario, consisting of a network of multiple queues. The aim was to optimize the bed allocation based on a specific objective. By analyzing a real-world hospital case study, we compared the expected metrics derived from queueing theory (assuming static arrival rates at each service department) to simulation results with dynamic arrival rates. Our finding revealed that for an M(t)/M/k queue with dynamic arrival rates exhibiting periodicity and low variation (e.g., 10%), simulation metrics (such as mean queue length, mean service time, etc.) closely aligh with those predicted by queueing theory with static arrival rates. This insight has significant practical implications, offering greater flexibility in selecting the most appropriate modeling approach for addressing similar real-world problems.

Furthermore, we applied a simulated annealing (SA) algorithm to find an optimal solution for bed allocation across the hospital's service departments. Our models and analyses provided a framework for optimizing bed distribution within these departments. We also conducted sensitivity analysis to assess how to allocate beds at each department in the context of additional bed purchases or achieving specific hospital goals. Our framework can serve as a decision-making support system, aiding hospital management in allocating resources based on various operational strategies.

5.2 Future Work

In our real-world case study, we formulated the optimization problem with a specific objective and constraints. However, there may be more potential studies extended from our models.

Objective Transformation

For the optimization problem, we established a single objective: minimizing the total nonzero wait time across the hospital's service departments. However, the weighting of the nonzero wait time at each department is debatable. One approach could be to weight each department's nonzero wait time according to the proportion of the arrivals relative to the entire hospital's arrivals. Another option could be to incorporate the importance of each department (determined through various means). With these objective changes, the optimal solution within the hospital may vary accordingly. Considering that alternative optimization problems may arise, further analysis of these alternatives would be necessary. However, the optimal solutions derived in this investigation can always serve as backups for the hospital bed allocation in these alternative scenarios.

Additionally, we could introduce more objectives into the optimization problem. For instance, integrating the objective of minimizing total utilization across service departments would complicate the problem, as it would involve two objectives, and finding an optimal solution that minimizes both may not be feasible. As we mentioned, this alternative problem formulation deserves deeper analysis.

Arrival Pattern Change

In our study, we assumed a sinusoidal arrival rate with low fluctuation for each service department. However, other arrival patterns are possible, such as a high variation sinusoidal function, a different functional form, or peak-hour arrivals with low off-peak

arrivals. These alternative patterns may yield different comparison results than those observed in our case study.

When the arrival patterns deviate significantly from the assumptions of static arrival rates used in queueing theory, the comparison between queueing theory and simulation results can diverge. In such cases, simulation modeling may be a more suitable approach for analyzing system performance. Therefore, further research is needed to explore the impact of various arrival patterns on system performance and the applicability of queueing theory versus simulation modeling.

Identify New Factors for Optimization Development

This study provides a foundation for future research on hospital operations. While we focused on a specific configuration, other hospital layouts may be beyond those considered in this study. For example, some hospitals feature a design pattern with wards dedicated to specific medical specialties (primary wards) and wards that can accommodate patients from differing medical specializations (non-primary wards). In this pattern, patients initially arrive at their primary ward. If beds are unavailable, some patients are transferred to non-primary wards base on certain rules. Moreover, we might consider scenarios where some patients start in non-primary wards and are later moved to their corresponding primary wards. These more complex cases require further analysis during the studying.

Additionally, we may consider the impact of the availability of other resources, such as nurses and cleaning crews, which significantly affect the service time. For instance, increasing staff during peak hours of the day can effectively reduce nonzero wait times, whereas reducing staff during non-busy periods may have minimal or no impact on nonzero wait times. Also, training more nurses to serve across multiple service departments and ensuring properly staffing across these departments can further decrease

nonzero wait times. Incorporating these additional considerations into the mathematical model may require modifications to the methodologies applied in this investigation.

Furthermore, hospital operational policies can influence system performance as well. For instance, implementing an early-discharge policy can result in reduced wait times[22]. These scenarios necessitate more complex modeling approaches, such as simulation, to accurately capture the dynamic nature of hospital operations. Further research should delve deeper into these areas, aiming to develop more comprehensive and accurate models for optimizing hospital operations.

In summary, this study serves as a foundation for further analysis in operations research for a network of queues. We have established that queueing theory is a particularly valuable for studying phenomena with static arrival rates, while simulation analysis excels in examining dynamic arrival rates cases. Given that real-world scenarios often involve complex dynamic arrival rates, this underscores the importance of simulation model in analyzing hospital operations metrics.

APPENDIX A

PYTHON IMPLEMENTATION OF QUEUEING THEORY FOR PERFORMANCE ANALYSIS

queueing.py:

```python
import numpy as np
import math

def erlangB_lostpro(s, a):
    n = 0
    B = 1
    while n < s:
        n += 1
        B = a * B / (n + a * B)
    return B

def erlangC(s, a):
    X = erlangB_lostpro(s, a)
    return s * X / (s - a * (1 - X)) if a / s < 1 else 1

def den(s, a):
    num = a**s
    den = math.factorial(s)*(1 - a/s)
    return num/den

def calculate_metrics(param):
    lamb = param["arrival_rate"] * (1 - param["proportion"])
    a = lamb / param["service_rate"]
    s = param["num_beds"]
    ErlangC_value = erlangC(s, a)
    rho = a / s
    P0 = ErlangC_value / den(s, a)    # first get erlangC value, then derive P0
```

```
    # Theoretical values
    W_time = ErlangC_value / ((1 - rho) * s * param["service_rate"])
    W_time_nonzero = 1 / ((1 - rho) * s * param["service_rate"])
    scale_param = W_time_nonzero
    W_size = lamb * W_time
    S_size = W_size + a
    Service_time = 1 / param["service_rate"]
    Sojourn_time = W_time + Service_time
    Throughput = lamb
    return {
        "a": a,
        "ErlangC_value": ErlangC_value,
        "P0": P0,
        "W_size": W_size,
        "S_size": S_size,
        "W_time": W_time,
        "W_time_nonzero": W_time_nonzero,
        "scale_param": scale_param,
        "Service_time": Service_time,
        "Sojourn_time": Sojourn_time,
        "rho": rho,
        "lamb": lamb,
        "Throughput": Throughput
    }

def main():
    # 630-24.5/24
    params_streams = [
        {"arrival_rate": 24.5 / 24, "service_rate": 1 / (2.37 * 24), "num_beds
    ": 65, "proportion": 0.0, "stream_id": 1},
        {"arrival_rate": 37 / 24, "service_rate": 1 / (3.02 * 24), "num_beds":
     119, "proportion": 0.0, "stream_id": 2},
```

```
        {"arrival_rate": 26 / 24, "service_rate": 1 / (4.09 * 24), "num_beds":
 115, "proportion": 0.0, "stream_id": 3},
        {"arrival_rate": 19 / 24, "service_rate": 1 / (2.89 * 24), "num_beds":
 62, "proportion": 0.0, "stream_id": 4},
        {"arrival_rate": 16.5 / 24, "service_rate": 1 / (3.27 * 24), "num_beds
": 62, "proportion": 0.0, "stream_id": 5},
        {"arrival_rate": 14.1 / 24, "service_rate": 1 / (3.51 * 24), "num_beds
": 57, "proportion": 0.0, "stream_id": 6},
        {"arrival_rate": 9.8 / 24, "service_rate": 1 / (5.56 * 24), "num_beds"
: 65, "proportion": 0.0, "stream_id": 7},
        {"arrival_rate": 16.3 / 24, "service_rate": 1 / (4.63 * 24), "num_beds
": 85, "proportion": 0.0, "stream_id": 8},
    ]

    for param in params_streams:
        metrics = calculate_metrics(param)
        print(f'PatientService Stream {param["stream_id"]} with Offered Load =
 {metrics["a"]}')
        print(f'PatientService Stream {param["stream_id"]} with Carried Load =
 {metrics["a"]}')
        print(f'PatientService Stream {param["stream_id"]} with ErlangC Value
= {metrics["ErlangC_value"]}')
        print(f'PatientService Stream {param["stream_id"]} with the Probablity
 of Zero Customer = {metrics["P0"]}')
        print(f'PatientService Stream {param["stream_id"]} with Mean Queue
Size = {metrics["W_size"]}')
        print(f'PatientService Stream {param["stream_id"]} with Mean Number of
 Patients in the System = {metrics["S_size"]}')
        print(f'PatientService Stream {param["stream_id"]} with Mean Wait Time
 = {metrics["W_time"]}')
        print(f'PatientService Stream {param["stream_id"]} with Mean Nonzero
Wait Time = {metrics["W_time_nonzero"]}')
```

```
76          print(f'PatientService Stream {param["stream_id"]} with scale_param
    for nonzero wait time  = {metrics["scale_param"]}')
77          print(f'PatientService Stream {param["stream_id"]} with Mean Service
    Time = {metrics["Service_time"]}')
78          print(f'PatientService Stream {param["stream_id"]} with Mean Sojourn
    Time = {metrics["Sojourn_time"]}')
79          print(f'PatientService Stream {param["stream_id"]} with Mean
    Utilization = {metrics["rho"]}')
80          print(f'PatientService Stream {param["stream_id"]} with Mean Arrival
    Rate = {metrics["lamb"]}')
81          print(f'PatientService Stream {param["stream_id"]} with Mean
    Throughput = {metrics["Throughput"]}')
82          print('-----------------------------------')
83
84  if __name__ == "__main__":
85      main()
```

Listing A.1: Queueing theory performance analysis

APPENDIX B

PYTHON SCRIPT FOR IDENTIFYING OPTIMAL WARMUP PERIOD

streams_nowarmup.py:

```
import json
import os
import simpy
import numpy as np
import seaborn as sns
import itertools
import random
import math
import matplotlib.pyplot as plt

class PatientService:
    def __init__(self, env, num_beds, servicetime):
        self.env = env
        self.bed = simpy.Resource(env, num_beds)
        self.servicetime = servicetime

    def serve(self, patient):
        yield self.env.timeout(np.random.exponential(scale=self.servicetime))

def patient(env, name, ps, num_beds, proportion, stream_id, all_data):
    arrive = env.now
    if random.random() < proportion:
        all_data["leaving_patients"][stream_id].append((arrive, stream_id))
        return

    if arrive > all_data["warm_up_period"]:
        all_data["queue_lengths"][stream_id].append((env.now, len(ps.bed.queue
    )))
```

```
        all_data["num_patients"][stream_id].append((env.now, len(ps.bed.queue)
    + len(ps.bed.users)))

    with ps.bed.request() as request:
        yield request
        wait = env.now - arrive

        if arrive > all_data["warm_up_period"]:
            all_data["waiting_times"][stream_id].append(wait)
            if wait > 0:
                all_data["nonzero_waiting_times"][stream_id].append(wait)

        service_start = env.now
        yield env.process(ps.serve(name))
        service_time = env.now - service_start
        if arrive > all_data["warm_up_period"]:
            all_data["service_times"][stream_id].append(service_time)
            all_data["system_times"][stream_id].append(wait + service_time)
            all_data["total_service_time"][stream_id] += service_time
            all_data["utilization"][stream_id].append((env.now, all_data["
    total_service_time"][stream_id] / (num_beds * (env.now - all_data["
    warm_up_period"]))))
            all_data["served"][stream_id] += 1
            all_data["throughput"][stream_id].append((env.now, all_data["
    served"][stream_id] / (env.now - all_data["warm_up_period"])))

# for dynamic arrival rates
def setup(env, base_arrival_rate, service_rate, num_beds, proportion,
    stream_id, all_data):
    patientservice = PatientService(env, num_beds, 1/service_rate)
    patient_count = itertools.count()

```

```
    max_arrival_rate = base_arrival_rate * 1.1  # Maximum rate considering the sinusoidal fluctuation

    while True:
        interarrival_time = np.random.exponential(scale=1/max_arrival_rate)
        yield env.timeout(interarrival_time)
        t = env.now
        current_arrival_rate = base_arrival_rate + 0.1 * base_arrival_rate * math.sin(math.pi * t / 12)
        if random.random() <= current_arrival_rate / max_arrival_rate:
            env.process(patient(env, f'Patient {next(patient_count)}', patientservice, num_beds, proportion, stream_id, all_data))

def run_simulation(params, warm_up_period, sim_time):
    all_data = {key: {i: [] for i in range(1, 9)} for key in [
        'leaving_patients', 'queue_lengths', 'num_patients', 'waiting_times',
        'nonzero_waiting_times', 'service_times', 'system_times', 'utilization', 'throughput']}
    all_data['warm_up_period'] = warm_up_period
    all_data['total_service_time'] = {i: 0 for i in range(1, 9)}
    all_data['served'] = {i: 0 for i in range(1, 9)}

    for param in params:
        random_seed = random.randint(40, 300)
        print(f'PatientService Stream {param["stream_id"]} with RANDOM_SEED = {random_seed}')
        random.seed(random_seed)
        env = simpy.Environment()
        env.process(setup(env, param["arrival_rate"], param["service_rate"], param["num_beds"], param["proportion"], param["stream_id"], all_data))
        env.run(until=sim_time)

```

```
81      return all_data
82
83  def save_data_to_json(all_data, filename='all_data.json'):
84      serializable_data = {}
85      for key, value in all_data.items():
86          if key == 'warm_up_period':
87              serializable_data[key] = value
88          else:
89              serializable_data[key] = {k: v for k, v in value.items()}
90      with open(filename, 'w') as f:
91          json.dump(serializable_data, f)
92
93  def load_data_from_json(filename='all_data.json'):
94      with open(filename, 'r') as f:
95          data = json.load(f)
96      deserialized_data = {}
97      for key, value in data.items():
98          if key == 'warm_up_period':
99              deserialized_data[key] = value
100         else:
101             deserialized_data[key] = {int(k): v for k, v in value.items()}
102     return deserialized_data
103
104 def plot_metric_over_time(metric_data, title, xlabel, ylabel, colors,
        stream_ids, output_file, average_line=None):
105     plt.figure(figsize=(10, 6))
106     for stream_id, data in metric_data.items():
107         if stream_id in stream_ids:
108             if isinstance(data, list) and data:
109                 try:
110                     times, values = zip(*data)
111                     plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
```

```
                    plt.plot(times, values, color=colors[(stream_id - 1)],
    label=f'Stream {stream_id}')
                    if average_line is not None:
                        plt.axhline(y=average_line[stream_id-1], color='r',
    linestyle='--', label=f'Average {ylabel} = {average_line[stream_id-1]:.2f}
    ')
                    plt.xlabel(xlabel)
                    plt.ylabel(ylabel)
                    plt.title(f'{title} for Stream {stream_id}')
                    plt.legend()
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list
    format.")
    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()

def plot_distribution(metric_data, title, xlabel, ylabel, colors, stream_ids,
    output_file):
    plt.figure(figsize=(10, 6))
    for stream_id, data in metric_data.items():
        if stream_id in stream_ids:
            if isinstance(data, list) and data:
                try:
                    mean_value = np.mean(data)
                    max_value = max(data)
                    print(f'Mean {ylabel} for Stream {stream_id}: {mean_value}
    ')
                    print(f'Max {ylabel} for Stream {stream_id}: {max_value}')
                    plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
```

```
                    sns.histplot(data, kde=True, color=colors[(stream_id - 1)
    ])
                    plt.xlabel(xlabel)
                    plt.ylabel('Frequency')
                    plt.title(f'{title} for Stream {stream_id}')
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list
    format.")
    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()

def calculate_average_metrics(all_data, params):
    average_L = []
    average_Lq = []
    for param in params:
        arrival_rate = param["arrival_rate"] * (1 - param["proportion"])
        service_rate = param["service_rate"]
        num_beds = param["num_beds"]
        rho = arrival_rate / (num_beds * service_rate)

        # Calculate Pw
        lambda_mu_ratio = arrival_rate / service_rate
        summation_terms = sum((lambda_mu_ratio ** n) / math.factorial(n) for n
     in range(num_beds))
        numerator_pw = (lambda_mu_ratio ** num_beds) / (math.factorial(
    num_beds) * (1 - rho))
        Pw = numerator_pw / (summation_terms + numerator_pw)

        # Calculate L and Lq
```

```
        L = (arrival_rate / service_rate) + (Pw * rho / (1 - rho))
        Lq = L - (arrival_rate / service_rate)

        print(f"Stream {param['stream_id']}: L = {L}, Lq = {Lq}")

        average_L.append(L)
        average_Lq.append(Lq)
    return average_L, average_Lq

def visualize_results(all_data, params, output_dir='nonhome_nonwarmup_streams_plots_1run'):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    average_L, average_Lq = calculate_average_metrics(all_data, params)

    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']
    plot_metric_over_time(all_data['utilization'], 'Utilization Over Time', 'Time', 'Utilization', colors, range(1, 5), os.path.join(output_dir, 'utilization_1_4.png'))
    plot_metric_over_time(all_data['utilization'], 'Utilization Over Time', 'Time', 'Utilization', colors, range(5, 9), os.path.join(output_dir, 'utilization_5_8.png'))
    plot_metric_over_time(all_data['queue_lengths'], 'Queue Length Over Time', 'Time', 'Queue Length', colors, range(1, 5), os.path.join(output_dir, 'queue_length_1_4.png'), average_line=average_Lq)
    plot_metric_over_time(all_data['queue_lengths'], 'Queue Length Over Time', 'Time', 'Queue Length', colors, range(5, 9), os.path.join(output_dir, 'queue_length_5_8.png'), average_line=average_Lq)
    plot_distribution(all_data['nonzero_waiting_times'], 'Nonzero Waiting Time Distribution', 'Nonzero Waiting Time', 'Nonzero Waiting Time', colors, range(1, 5), os.path.join(output_dir, 'nonzero_waiting_times_1_4.png'))
```

```
187     plot_distribution(all_data['nonzero_waiting_times'], 'Nonzero Waiting Time
         Distribution', 'Nonzero Waiting Time', 'Nonzero Waiting Time', colors,
        range(5, 9), os.path.join(output_dir, 'nonzero_waiting_times_5_8.png'))
188     plot_metric_over_time(all_data['num_patients'], 'Number of Patients Over
        Time', 'Time', 'Number of Patients', colors, range(1, 5), os.path.join(
        output_dir, 'num_patients_1_4.png'), average_line=average_L)
189     plot_metric_over_time(all_data['num_patients'], 'Number of Patients Over
        Time', 'Time', 'Number of Patients', colors, range(5, 9), os.path.join(
        output_dir, 'num_patients_5_8.png'), average_line=average_L)
190     plot_distribution(all_data['waiting_times'], 'Waiting Time Distribution',
        'Waiting Time', 'Waiting Time', colors, range(1, 5), os.path.join(
        output_dir, 'waiting_times_1_4.png'))
191     plot_distribution(all_data['waiting_times'], 'Waiting Time Distribution',
        'Waiting Time', 'Waiting Time', colors, range(5, 9), os.path.join(
        output_dir, 'waiting_times_5_8.png'))
192     plot_distribution(all_data['service_times'], 'Service Time Distribution',
        'Service Time', 'Service Time', colors, range(1, 5), os.path.join(
        output_dir, 'service_times_1_4.png'))
193     plot_distribution(all_data['service_times'], 'Service Time Distribution',
        'Service Time', 'Service Time', colors, range(5, 9), os.path.join(
        output_dir, 'service_times_5_8.png'))
194     plot_distribution(all_data['system_times'], 'System Time Distribution', '
        System Time', 'System Time', colors, range(1, 5), os.path.join(output_dir,
         'system_times_1_4.png'))
195     plot_distribution(all_data['system_times'], 'System Time Distribution', '
        System Time', 'System Time', colors, range(5, 9), os.path.join(output_dir,
         'system_times_5_8.png'))
196     plot_metric_over_time(all_data['throughput'], 'Throughput Over Time', '
        Time', 'Throughput', colors, range(1, 5), os.path.join(output_dir, '
        throughput_1_4.png'))
197     plot_metric_over_time(all_data['throughput'], 'Throughput Over Time', '
        Time', 'Throughput', colors, range(5, 9), os.path.join(output_dir, '
```

```
    throughput_5_8.png'))


def main():
    params = [
        {"arrival_rate": 24.5 / 24, "service_rate": 1 / (2.37 * 24), "num_beds": 65, "proportion": 0.0, "stream_id": 1},
        {"arrival_rate": 37 / 24, "service_rate": 1 / (3.02 * 24), "num_beds": 119, "proportion": 0.0, "stream_id": 2},
        {"arrival_rate": 26 / 24, "service_rate": 1 / (4.09 * 24), "num_beds": 115, "proportion": 0.0, "stream_id": 3},
        {"arrival_rate": 19 / 24, "service_rate": 1 / (2.89 * 24), "num_beds": 62, "proportion": 0.0, "stream_id": 4},
        {"arrival_rate": 16.5 / 24, "service_rate": 1 / (3.27 * 24), "num_beds": 62, "proportion": 0.0, "stream_id": 5},
        {"arrival_rate": 14.1 / 24, "service_rate": 1 / (3.51 * 24), "num_beds": 57, "proportion": 0.0, "stream_id": 6},
        {"arrival_rate": 9.8 / 24, "service_rate": 1 / (5.56 * 24), "num_beds": 65, "proportion": 0.0, "stream_id": 7},
        {"arrival_rate": 16.3 / 24, "service_rate": 1 / (4.63 * 24), "num_beds": 85, "proportion": 0.0, "stream_id": 8}
    ]
    warm_up_period = 0
    sim_time = 30000

    if not os.path.exists('all_data.json'):
        all_data = run_simulation(params, warm_up_period, sim_time)
        save_data_to_json(all_data)
    else:
        all_data = load_data_from_json()

    visualize_results(all_data, params)
```

```
221
222 if __name__ == "__main__":
223     main()
```

Listing B.1: Python script for identifying warmup

APPENDIX C

PYTHON IMPLEMENTATION OF METRIC CALCULATION FOR A SINGLE SIMULATION RUN

streams_nonhomo_warmup_run1.py:

```
import json
import os
import simpy
import numpy as np
import seaborn as sns
import itertools
import random
import math
import matplotlib.pyplot as plt

class PatientService:
    def __init__(self, env, num_beds, servicetime):
        self.env = env
        self.bed = simpy.Resource(env, num_beds)
        self.servicetime = servicetime

    def serve(self, patient):
        yield self.env.timeout(random.expovariate(1/self.servicetime))

def patient(env, name, ps, num_beds, proportion, stream_id, all_data):
    arrive = env.now

    # Only count arrivals after warm-up period
    if arrive > all_data["warm_up_period"]:
        all_data["total_arrivals"][stream_id] += 1  # Increment the count of
    arrivals
        all_data["arrival_rates"][stream_id].append((env.now, all_data["
```

```
total_arrivals"][stream_id]/(env.now - all_data["warm_up_period"])))

    if random.random() < proportion:
        all_data["leaving_patients"][stream_id].append((arrive, stream_id))
        return

    if arrive > all_data["warm_up_period"]:
        all_data["queue_lengths"][stream_id].append((env.now, len(ps.bed.queue
))))
        all_data["num_patients"][stream_id].append((env.now, len(ps.bed.queue)
     + len(ps.bed.users)))

    with ps.bed.request() as request:
        yield request
        wait = env.now - arrive

        if arrive > all_data["warm_up_period"]:
            all_data["wait_times"][stream_id].append(wait)
            if wait > 0:
                all_data["nonzero_wait_times"][stream_id].append(wait)

        service_start = env.now
        yield env.process(ps.serve(name))
        service_time = env.now - service_start
        if arrive > all_data["warm_up_period"]:
            all_data["service_times"][stream_id].append(service_time)
            all_data["sojourn_times"][stream_id].append(wait + service_time)
            all_data["total_service_time"][stream_id] += service_time
            all_data["utilization"][stream_id].append((env.now, all_data["
total_service_time"][stream_id] / (num_beds * (env.now - all_data["
warm_up_period"]))))
            all_data["served"][stream_id] += 1
```

```
            all_data["throughput"][stream_id].append((env.now, all_data["served"][stream_id] / (env.now - all_data["warm_up_period"])))

# for dynamic arrival rates
def setup(env, base_arrival_rate, service_rate, num_beds, proportion, stream_id, all_data):
    patientservice = PatientService(env, num_beds, 1/service_rate)
    patient_count = itertools.count()

    max_arrival_rate = base_arrival_rate * 1.1  # Maximum rate considering the sinusoidal fluctuation

    while True:
        interarrival_time = random.expovariate(max_arrival_rate)
        yield env.timeout(interarrival_time)

        t = env.now
        current_arrival_rate = base_arrival_rate + 0.1 * base_arrival_rate * math.sin(math.pi * t / 12)

        if random.random() <= current_arrival_rate / max_arrival_rate:
            env.process(patient(env, f'Patient {next(patient_count)}', patientservice, num_beds, proportion, stream_id, all_data))

def run_simulation(params, warm_up_period, sim_time):
    all_data = {key: {i: [] for i in range(1, 9)} for key in [
        'arrival_rates', 'leaving_patients', 'queue_lengths', 'num_patients', 'wait_times',
        'nonzero_wait_times', 'service_times', 'sojourn_times', 'utilization', 'throughput']}
    all_data['warm_up_period'] = warm_up_period
    all_data['total_arrivals'] = {i: 0 for i in range(1, 9)}  # New key to
```

```
    track arrivals
    all_data['total_service_time'] = {i: 0 for i in range(1, 9)}     # to
    calculate utilization
    all_data['served'] = {i: 0 for i in range(1, 9)}                 # to
    calculate throughput

    for param in params:
        random_seed = random.randint(40, 300)
        print(f'PatientService Stream {param["stream_id"]} with RANDOM_SEED =
    {random_seed}')
        random.seed(random_seed)
        env = simpy.Environment()
        env.process(setup(env, param["arrival_rate"], param["service_rate"],
    param["num_beds"], param["proportion"], param["stream_id"], all_data))
        env.run(until=sim_time)

    return all_data

def save_data_to_json(all_data, filename='all_data.json'):
    serializable_data = {}
    for key, value in all_data.items():
        if key == 'warm_up_period':
            serializable_data[key] = value
        else:
            serializable_data[key] = {k: v for k, v in value.items()}
    with open(filename, 'w') as f:
        json.dump(serializable_data, f)

def load_data_from_json(filename='all_data.json'):
    with open(filename, 'r') as f:
        data = json.load(f)
    deserialized_data = {}
```

```
    for key, value in data.items():
        if key == 'warm_up_period':
            deserialized_data[key] = value
        else:
            deserialized_data[key] = {int(k): v for k, v in value.items()}
    return deserialized_data

def plot_metric_over_time(metric_data, title, xlabel, ylabel, colors, stream_ids, output_file, average_line=None):
    plt.figure(figsize=(10, 6))
    for stream_id, data in metric_data.items():
        if stream_id in stream_ids:
            if isinstance(data, list) and data:
                try:
                    times, values = zip(*data)
                    plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
                    plt.plot(times, values, color=colors[(stream_id - 1)], label=f'Stream {stream_id}')
                    if average_line is not None:
                        plt.axhline(y=average_line[stream_id-1], color='r', linestyle='--', label=f'Average {ylabel} (Theoretical)= {average_line[stream_id-1]:.2f}')
                    plt.xlabel(xlabel)
                    plt.ylabel(ylabel)
                    plt.title(f'{title} for Stream {stream_id}')
                    plt.legend()
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list format.")
    plt.tight_layout()
```

```
    plt.savefig(output_file)
    plt.show()

def plot_distribution(metric_data, title, xlabel, ylabel, colors, stream_ids,
    output_file):
    plt.figure(figsize=(10, 6))
    for stream_id, data in metric_data.items():
        if stream_id in stream_ids:
            if isinstance(data, list) and data:
                try:
                    mean_value = np.mean(data)
                    max_value = max(data)
                    print(f'Mean {ylabel} for Stream {stream_id} (Simulation):
     {mean_value}')
                    print(f'Max {ylabel} for Stream {stream_id} (Simulation):
    {max_value}')
                    plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
                    sns.histplot(data, kde=True, color=colors[(stream_id - 1)
    ])
                    plt.xlabel(xlabel)
                    plt.ylabel('Frequency')
                    plt.title(f'{title} for Stream {stream_id}')
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list
    format.")
    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()

def calculate_average_metrics(all_data, params):     # from queueing theory,
```

```
    dashlines
    average_L = []
    average_Lq = []
    for param in params:
        arrival_rate = param["arrival_rate"] * (1 - param["proportion"])
        service_rate = param["service_rate"]
        num_beds = param["num_beds"]
        rho = arrival_rate / (num_beds * service_rate)

        # Calculate Pw
        lambda_mu_ratio = arrival_rate / service_rate
        summation_terms = sum((lambda_mu_ratio ** n) / math.factorial(n) for n
    in range(num_beds))
        numerator_pw = (lambda_mu_ratio ** num_beds) / (math.factorial(
    num_beds) * (1 - rho))
        Pw = numerator_pw / (summation_terms + numerator_pw)

        # Calculate L and Lq
        L = (arrival_rate / service_rate) + (Pw * rho / (1 - rho))
        Lq = L - (arrival_rate / service_rate)

        print(f"In theory, Stream {param['stream_id']}: L = {L}, Lq = {Lq}")

        average_L.append(L)
        average_Lq.append(Lq)
    return average_L, average_Lq

def visualize_results(all_data, params, output_dir='
    nonhome_warmup_streams_plots_1run'):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

```

```
    average_L, average_Lq = calculate_average_metrics(all_data, params)

    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']

    plot_metric_over_time(all_data['queue_lengths'], 'Queue Length Over Time',
     'Time', 'Queue Length', colors, range(1, 5), os.path.join(output_dir, '
queue_length_1_4.png'), average_line=average_Lq)
    plot_metric_over_time(all_data['queue_lengths'], 'Queue Length Over Time',
     'Time', 'Queue Length', colors, range(5, 9), os.path.join(output_dir, '
queue_length_5_8.png'), average_line=average_Lq)
    plot_metric_over_time(all_data['num_patients'], 'Number of Patients Over
Time', 'Time', 'Number of Patients', colors, range(1, 5), os.path.join(
output_dir, 'num_patients_1_4.png'), average_line=average_L)
    plot_metric_over_time(all_data['num_patients'], 'Number of Patients Over
Time', 'Time', 'Number of Patients', colors, range(5, 9), os.path.join(
output_dir, 'num_patients_5_8.png'), average_line=average_L)
    plot_distribution(all_data['wait_times'], 'Wait Time Distribution', 'Wait
Time', 'Wait Time', colors, range(1, 5), os.path.join(output_dir, '
wait_times_1_4.png'))
    plot_distribution(all_data['wait_times'], 'Wait Time Distribution', 'Wait
Time', 'Wait Time', colors, range(5, 9), os.path.join(output_dir, '
wait_times_5_8.png'))
    plot_distribution(all_data['nonzero_wait_times'], 'Nonzero Wait Time
Distribution', 'Nonzero Wait Time', 'Nonzero Wait Time', colors, range(1,
5), os.path.join(output_dir, 'nonzero_wait_times_1_4.png'))
    plot_distribution(all_data['nonzero_wait_times'], 'Nonzero Wait Time
Distribution', 'Nonzero Wait Time', 'Nonzero Wait Time', colors, range(5,
9), os.path.join(output_dir, 'nonzero_wait_times_5_8.png'))
    plot_distribution(all_data['service_times'], 'Service Time Distribution',
'Service Time', 'Service Time', colors, range(1, 5), os.path.join(
output_dir, 'service_times_1_4.png'))
    plot_distribution(all_data['service_times'], 'Service Time Distribution',
```

```
    'Service Time', 'Service Time', colors, range(5, 9), os.path.join(
    output_dir, 'service_times_5_8.png'))
202     plot_distribution(all_data['sojourn_times'], 'Sojourn Time Distribution',
    'Sojourn Time', 'Sojourn Time', colors, range(1, 5), os.path.join(
    output_dir, 'sojourn_times_1_4.png'))
203     plot_distribution(all_data['sojourn_times'], 'Sojourn Time Distribution',
    'Sojourn Time', 'Sojourn Time', colors, range(5, 9), os.path.join(
    output_dir, 'sojourn_times_5_8.png'))
204     plot_metric_over_time(all_data['utilization'], 'Utilization Over Time', '
    Time', 'Utilization', colors, range(1, 5), os.path.join(output_dir, '
    utilization_1_4.png'))
205     plot_metric_over_time(all_data['utilization'], 'Utilization Over Time', '
    Time', 'Utilization', colors, range(5, 9), os.path.join(output_dir, '
    utilization_5_8.png'))
206     plot_metric_over_time(all_data['arrival_rates'], 'Arrival Rate Over Time',
     'Time', 'Arrival Rate', colors, range(1, 5), os.path.join(output_dir, '
    arrival_rates_1_4.png'))
207     plot_metric_over_time(all_data['arrival_rates'], 'Arrival Rate Over Time',
     'Time', 'Arrival Rate', colors, range(5, 9), os.path.join(output_dir, '
    arrival_rates_5_8.png'))
208     plot_metric_over_time(all_data['throughput'], 'Throughput Over Time', '
    Time', 'Throughput', colors, range(1, 5), os.path.join(output_dir, '
    throughput_1_4.png'))
209     plot_metric_over_time(all_data['throughput'], 'Throughput Over Time', '
    Time', 'Throughput', colors, range(5, 9), os.path.join(output_dir, '
    throughput_5_8.png'))
210
211
212 def main():
213     params = [
214         {"arrival_rate": 24.5 / 24, "service_rate": 1 / (2.37 * 24), "num_beds
    ": 65, "proportion": 0.0, "stream_id": 1},
```

```
215         {"arrival_rate": 37 / 24, "service_rate": 1 / (3.02 * 24), "num_beds":
    119, "proportion": 0.0, "stream_id": 2},
216         {"arrival_rate": 26 / 24, "service_rate": 1 / (4.09 * 24), "num_beds":
    115, "proportion": 0.0, "stream_id": 3},
217         {"arrival_rate": 19 / 24, "service_rate": 1 / (2.89 * 24), "num_beds":
    62, "proportion": 0.0, "stream_id": 4},
218         {"arrival_rate": 16.5 / 24, "service_rate": 1 / (3.27 * 24), "num_beds
   ": 62, "proportion": 0.0, "stream_id": 5},
219         {"arrival_rate": 14.1 / 24, "service_rate": 1 / (3.51 * 24), "num_beds
   ": 57, "proportion": 0.0, "stream_id": 6},
220         {"arrival_rate": 9.8 / 24, "service_rate": 1 / (5.56 * 24), "num_beds"
   : 65, "proportion": 0.0, "stream_id": 7},
221         {"arrival_rate": 16.3 / 24, "service_rate": 1 / (4.63 * 24), "num_beds
   ": 85, "proportion": 0.0, "stream_id": 8}
222     ]
223     warm_up_period = 10000
224     sim_time = 310000
225
226     if not os.path.exists('all_data.json'):
227         all_data = run_simulation(params, warm_up_period, sim_time)
228         save_data_to_json(all_data)
229     else:
230         all_data = load_data_from_json()
231
232     visualize_results(all_data, params)
233     # calculate_arrival_rate(all_data, sim_time)
234
235 if __name__ == "__main__":
236     main()
237
238 % \end{verbatim}
```

Listing C.1: Metric calculation for a single run

APPENDIX D

PYTHON SCRIPT FOR GOODNESS-OF-FIT TESTING OF SERVICE TIME DISTRIBUTIONS

streams_goodness_of_fit_test_service_times.py:

```
import json
import os
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import kstest, expon

# The all_data.json file for this analysis is generated from
# streams_nonhomo_warmup_run1.py (see Appendix C)

def load_data_from_json(filename='all_data.json'):
    with open(filename, 'r') as f:
        data = json.load(f)
    deserialized_data = {}
    for key, value in data.items():
        if key == 'warm_up_period':
            deserialized_data[key] = value
        else:
            deserialized_data[key] = {int(k): v for k, v in value.items()}
    return deserialized_data


def perform_exponential_ks_test(service_times):
    flattened_service_times = np.concatenate([np.array(sublist) for sublist in
    service_times.values()])

    # Allow non-negative values, including zeros
```

```
    valid_service_times = flattened_service_times[flattened_service_times >= 0]

    if len(valid_service_times) == 0:
        raise ValueError("No valid data available for fitting the Exponential distribution.")

    # Fit the exponential distribution to the data
    loc, scale = expon.fit(valid_service_times, floc=0)

    # Perform K-S test
    d, p_value = kstest(valid_service_times, 'expon', args=(loc, scale))

    return d, p_value, loc, scale


def plot_multiple_cdfs(stream_ids, all_service_times, all_scales, colors, output_file):
    from scipy.stats import expon
    plt.figure(figsize=(12, 10))

    for i, stream_id in enumerate(stream_ids, start=1):
        service_times = all_service_times[stream_id]
        scale = all_scales[stream_id]  # No shape parameter for Exponential distribution

        sorted_data = np.sort(service_times)
        yvals = np.arange(len(sorted_data)) / float(len(sorted_data))

        # Empirical CDF
        plt.subplot(2, 2, i)
        plt.step(sorted_data, yvals, label="Empirical CDF", color=colors[(
```

```
    stream_id - 1) % len(colors)])

        # Theoretical Exponential CDF
        plt.plot(sorted_data, expon.cdf(sorted_data, loc=0, scale=scale),
                 label=f"Theoretical Exponential CDF", color='red', lw=2)

        plt.title(f'CDF of Service Times (Stream {stream_id})')
        plt.xlabel('Service Time')
        plt.ylabel('CDF')
        plt.legend()
        plt.grid(True)

    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()


def plot_distribution(metric_data, title, xlabel, ylabel, colors, stream_ids,
    output_file):
    plt.figure(figsize=(12, 10))
    for stream_id, data in metric_data.items():
        if stream_id in stream_ids:
            if isinstance(data, list) and data:
                try:
                    mean_value = np.mean(data)
                    max_value = max(data)
                    print(f'Mean {ylabel} for Stream {stream_id}: {mean_value}
    ')
                    print(f'Max {ylabel} for Stream {stream_id}: {max_value}')
                    plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
                    # Plot histogram and KDE with clipping at 0
                    sns.histplot(data, kde=False, color=colors[(stream_id - 1)
```

```
    ], stat="density", alpha=0.6)
                    sns.kdeplot(data, color='black', lw=2, clip=(0, np.inf))
    # Ensure no values below 0
                    plt.axvline(mean_value, color='r', linestyle='--', label=f
    'Mean = {mean_value:.2f}')
                    plt.xlabel(xlabel)
                    plt.ylabel(ylabel)
                    plt.title(f'{title} for Stream {stream_id}')
                    plt.legend()
                    plt.grid(True)
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list
    format.")
    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()


def main():
    # Create directory for plots
    output_dir = 'streams_service_times_expon_test'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    all_data = load_data_from_json('all_data.json')

    # Store scale parameters for each stream
    all_scales = {}

    # Perform Exponential K-S test on service time data for each stream
```

```
    for stream_id, service_times in all_data['service_times'].items():
        d, p_value, loc, scale = perform_exponential_ks_test({stream_id: service_times})
        all_scales[stream_id] = scale
        print(f'Stream {stream_id}: D = {d}, p-value = {p_value}, scale = {scale}')

    # Define colors for streams
    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']

    # Plot and save CDFs for streams 1-4 and 5-8, with different colors for each stream
    plot_multiple_cdfs(range(1, 5), all_data['service_times'], all_scales,
                       colors, os.path.join(output_dir, 'cdf_service_times_1_4.png'))
    plot_multiple_cdfs(range(5, 9), all_data['service_times'], all_scales,
                       colors, os.path.join(output_dir, 'cdf_service_times_5_8.png'))

    # Plot and save service time distributions for streams 1-4 and 5-8
    plot_distribution(all_data['service_times'], 'Service Time Distribution',
                      'Service Time', 'Density', colors, range(1, 5),
                      os.path.join(output_dir, 'service_time_distribution_1_4.png'))
    plot_distribution(all_data['service_times'], 'Service Time Distribution',
                      'Service Time', 'Density', colors, range(5, 9),
                      os.path.join(output_dir, 'service_time_distribution_5_8.png'))

if __name__ == "__main__":
    main()
```

Listing D.1: Goodness-of-Fit testing of service time distributions

APPENDIX E

PYTHON CODE FOR STATISTICAL INFERENCE OF SIMULATION RESULTS

streams_nonhomo_warmup_modular.py:

```
import simpy
import numpy as np
import random
import itertools
import math
import scipy.stats
import matplotlib.pyplot as plt
import seaborn as sns
import multiprocessing as mp
import json
import time
import logging
import os

class PatientService:
    def __init__(self, env, num_beds, servicetime):
        self.env = env
        self.bed = simpy.Resource(env, num_beds)
        self.servicetime = servicetime

    def serve(self, patient):
        yield self.env.timeout(np.random.exponential(scale=self.servicetime))

def patient(env, name, ps, num_beds, proportion, stream_id, results):
    arrive = env.now

    # Only track arrivals after the warm-up period
    if arrive > results["WARM_UP_PERIOD"]:
```

```
29        results["total_arrivals"][stream_id] += 1  # Increment the count of
    arrivals
30
31     if random.random() < proportion:
32         results["leaving_patients"][stream_id].append((arrive, stream_id))
33         return
34
35     if arrive > results["WARM_UP_PERIOD"]:
36         results["queue_lengths"][stream_id].append((env.now, len(ps.bed.queue)
    ))
37         results["num_patients"][stream_id].append((env.now, len(ps.bed.queue)
    + len(ps.bed.users)))
38
39     with ps.bed.request() as request:
40         yield request
41         wait = env.now - arrive
42
43         if arrive > results["WARM_UP_PERIOD"]:
44             results["wait_times"][stream_id].append(wait)
45             if wait > 0:
46                 results["nonzero_wait_times"][stream_id].append(wait)
47
48         service_start = env.now
49         yield env.process(ps.serve(name))
50         service_time = env.now - service_start
51         if arrive > results["WARM_UP_PERIOD"]:
52             results["service_times"][stream_id].append(service_time)
53             results["sojourn_times"][stream_id].append(wait + service_time)
54             results["total_service_time"][stream_id] += service_time
55             utilization = results["total_service_time"][stream_id] / (num_beds
     * (env.now - results["WARM_UP_PERIOD"]))
56             results["utilization_over_time"][stream_id].append((env.now,
```

```
    utilization))
57              results["served"][stream_id] += 1
58              results["throughput"][stream_id].append((env.now, results["served"
    ][stream_id] / (env.now - results["WARM_UP_PERIOD"])))
59
60  # for dynamic arrival rates with sinusoidal function
61  def setup(env, base_arrival_rate, service_rate, num_beds, proportion,
    stream_id, results):
62      patientservice = PatientService(env, num_beds, 1 / service_rate)
63      patient_count = itertools.count()
64
65      max_arrival_rate = base_arrival_rate * 2  # Maximum rate considering the
    sinusoidal fluctuation
66
67      while True:
68          interarrival_time = np.random.exponential(scale=1 / max_arrival_rate)
69          yield env.timeout(interarrival_time)
70
71          t = env.now
72          current_arrival_rate = base_arrival_rate + 1 * base_arrival_rate *
    math.sin(math.pi * t / 12)
73
74          if random.random() <= current_arrival_rate / max_arrival_rate:
75              env.process(patient(env, f'Patient {next(patient_count)}',
    patientservice, num_beds, proportion, stream_id, results))
76
77  def run_simulation(run, params, whole_run_leaving_patients, WARM_UP_PERIOD
    =10000, SIM_TIME=30000):
78      results = {
79          "WARM_UP_PERIOD": WARM_UP_PERIOD,
80          "total_arrivals": {i: 0 for i in range(1, 9)},  # Track total arrivals
     after warm-up
```

```
81
82          "leaving_patients": {i: [] for i in range(1, 9)},
83          "queue_lengths": {i: [] for i in range(1, 9)},
84          "num_patients": {i: [] for i in range(1, 9)},
85          "wait_times": {i: [] for i in range(1, 9)},
86          "nonzero_wait_times": {i: [] for i in range(1, 9)},
87          "service_times": {i: [] for i in range(1, 9)},
88          "sojourn_times": {i: [] for i in range(1, 9)},
89          "total_service_time": {i: 0 for i in range(1, 9)},
90
91          "utilization_over_time": {i: [] for i in range(1, 9)},
92          "served": {i: 0 for i in range(1, 9)},
93          "throughput": {i: [] for i in range(1, 9)},
94
95      }
96      for param in params:
97          env = simpy.Environment()
98          env.process(setup(env, param["arrival_rate"], param["service_rate"],
    param["num_beds"], param["proportion"], param["stream_id"], results))
99          env.run(until=SIM_TIME)
100
101     # for the tuples, get the numbers
102     mean_queue_lengths = {i: np.mean([q[1] for q in results["queue_lengths"][i
    ]]) for i in range(1, 9) if results["queue_lengths"][i]}
103     mean_num_patients = {i: np.mean([n[1] for n in results["num_patients"][i
    ]]) for i in range(1, 9) if results["num_patients"][i]}
104     mean_service_times = {i: np.mean(results["service_times"][i]) for i in
    range(1, 9) if results["service_times"][i]}
105     mean_sojourn_times = {i: np.mean(results["sojourn_times"][i]) for i in
    range(1, 9) if results["sojourn_times"][i]}
106     mean_throughput = {i: np.mean([t[1] for t in results["throughput"][i]])
    for i in range(1, 9) if results["throughput"][i]}
```

```
    mean_wait_times = {i: np.mean(results["wait_times"][i]) for i in range(1, 9) if results["wait_times"][i]}
    mean_nonzero_wait_times = {i: np.mean(results["nonzero_wait_times"][i]) for i in range(1, 9) if results["nonzero_wait_times"][i]}
    last_utilization = {i: results["utilization_over_time"][i][-1][1] for i in range(1, 9) if results["utilization_over_time"][i]}

    # Calculate the arrival rates after warm-up
    mean_arrival_rates = {i: results["total_arrivals"][i] / (SIM_TIME - WARM_UP_PERIOD) for i in range(1, 9)}

    whole_run_leaving_patients[run] = results["leaving_patients"]

    # each run has the following metrics
    return results["leaving_patients"], mean_queue_lengths, mean_num_patients, mean_wait_times, mean_nonzero_wait_times, mean_service_times, mean_sojourn_times, last_utilization, mean_throughput, mean_arrival_rates

def calculate_confidence_interval(data):
    confidence_level = 0.95
    degrees_freedom = len(data) - 1
    sample_mean = np.mean(data)
    sample_standard_error = scipy.stats.sem(data)
    confidence_interval = scipy.stats.t.interval(confidence_level, degrees_freedom, loc=sample_mean, scale=sample_standard_error)
    return confidence_interval

def main():
    params = [
        {"arrival_rate": 24.5 / 24, "service_rate": 1 / (2.37 * 24), "num_beds": 65, "proportion": 0.0, "stream_id": 1},
        {"arrival_rate": 37 / 24, "service_rate": 1 / (3.02 * 24), "num_beds":
```

```
    119, "proportion": 0.0, "stream_id": 2},
131         {"arrival_rate": 26 / 24, "service_rate": 1 / (4.09 * 24), "num_beds":
    115, "proportion": 0.0, "stream_id": 3},
132         {"arrival_rate": 19 / 24, "service_rate": 1 / (2.89 * 24), "num_beds":
    62, "proportion": 0.0, "stream_id": 4},
133         {"arrival_rate": 16.5 / 24, "service_rate": 1 / (3.27 * 24), "num_beds
"": 62, "proportion": 0.0, "stream_id": 5},
134         {"arrival_rate": 14.1 / 24, "service_rate": 1 / (3.51 * 24), "num_beds
"": 57, "proportion": 0.0, "stream_id": 6},
135         {"arrival_rate": 9.8 / 24, "service_rate": 1 / (5.56 * 24), "num_beds"
: 65, "proportion": 0.0, "stream_id": 7},
136         {"arrival_rate": 16.3 / 24, "service_rate": 1 / (4.63 * 24), "num_beds
"": 85, "proportion": 0.0, "stream_id": 8},
137     ]
138     num_runs = 50    # 50
139     warm_up_period = 10000
140     sim_time = 30000
141 
142     whole_run_leaving_patients = {run: {i: [] for i in range(1, 9)} for run in
     range(1, num_runs + 1)}
143 
144     aggregated_mean_queue_lengths = {i: [] for i in range(1, 9)}
145     aggregated_mean_num_patients = {i: [] for i in range(1, 9)}
146     aggregated_mean_service_times = {i: [] for i in range(1, 9)}
147     aggregated_mean_sojourn_times = {i: [] for i in range(1, 9)}
148     aggregated_mean_throughput = {i: [] for i in range(1, 9)}
149     aggregated_mean_wait_times = {i: [] for i in range(1, 9)}
150     aggregated_mean_nonzero_wait_times = {i: [] for i in range(1, 9)}
151     aggregated_last_utilization = {i: [] for i in range(1, 9)}
152     aggregated_mean_arrival_rates = {i: [] for i in range(1, 9)}
153 
154     start_time = time.time()
```

```
155     pool = mp.Pool(mp.cpu_count())
156     results = pool.starmap(run_simulation, [(run, params,
    whole_run_leaving_patients, warm_up_period, sim_time) for run in range(1,
    num_runs + 1)])
157     pool.close()
158     pool.join()
159
160     for run, (lps, mql, mnp, mwt, mnwt, mst, msojt, lu, mt, mar) in enumerate(
    results, 1):
161         for i in range(1, 9):
162             if i in mql and mql[i]:
163                 aggregated_mean_queue_lengths[i].append(mql[i])
164             if i in mnp and mnp[i]:
165                 aggregated_mean_num_patients[i].append(mnp[i])
166             if i in mwt and mwt[i]:
167                 aggregated_mean_wait_times[i].append(mwt[i])
168             if i in mnwt and mnwt[i]:
169                 aggregated_mean_nonzero_wait_times[i].append(mnwt[i])
170             if i in mst and mst[i]:
171                 aggregated_mean_service_times[i].append(mst[i])
172             if i in msojt and msojt[i]:
173                 aggregated_mean_sojourn_times[i].append(msojt[i])
174             if i in lu and lu[i]:
175                 aggregated_last_utilization[i].append(lu[i])
176             if i in mt and mt[i]:
177                 aggregated_mean_throughput[i].append(mt[i])
178             if i in mar and mar[i]:
179                 aggregated_mean_arrival_rates[i].append(mar[i])
180
181     end_time = time.time()
182     print(f"Total execution time: {end_time - start_time} seconds")
183
```

```
    # Calculate confidence intervals for all metrics
    confidence_intervals_queue_length = {i: calculate_confidence_interval(
aggregated_mean_queue_lengths[i]) for i in range(1, 9)}
    confidence_intervals_num_patient = {i: calculate_confidence_interval(
aggregated_mean_num_patients[i]) for i in range(1, 9)}
    confidence_intervals_service_time = {i: calculate_confidence_interval(
aggregated_mean_service_times[i]) for i in range(1, 9)}
    confidence_intervals_sojourn_time = {i: calculate_confidence_interval(
aggregated_mean_sojourn_times[i]) for i in range(1, 9)}
    confidence_intervals_throughput = {i: calculate_confidence_interval(
aggregated_mean_throughput[i]) for i in range(1, 9)}
    confidence_intervals_wait_time = {i: calculate_confidence_interval(
aggregated_mean_wait_times[i]) for i in range(1, 9)}
    confidence_intervals_nonzero_wait_time = {i: calculate_confidence_interval
(aggregated_mean_nonzero_wait_times[i]) for i in range(1, 9)}
    confidence_intervals_utilization = {i: calculate_confidence_interval(
aggregated_last_utilization[i]) for i in range(1, 9)}
    confidence_intervals_arrival_rate = {i: calculate_confidence_interval(
aggregated_mean_arrival_rates[i]) for i in range(1, 9)}

    # Print mean values and confidence intervals for all metrics
    mean_values_queue_length = {i: np.mean(aggregated_mean_queue_lengths[i])
for i in range(1, 9)}
    mean_values_num_patient = {i: np.mean(aggregated_mean_num_patients[i]) for
 i in range(1, 9)}
    mean_values_service_time = {i: np.mean(aggregated_mean_service_times[i])
for i in range(1, 9)}
    mean_values_sojourn_time = {i: np.mean(aggregated_mean_sojourn_times[i])
for i in range(1, 9)}
    mean_values_throughput = {i: np.mean(aggregated_mean_throughput[i]) for i
in range(1, 9)}
    mean_values_wait_time = {i: np.mean(aggregated_mean_wait_times[i]) for i
```

```
in range(1, 9)}
202  mean_values_nonzero_wait_time = {i: np.mean(
aggregated_mean_nonzero_wait_times[i]) for i in range(1, 9)}
203  mean_values_utilization = {i: np.mean(aggregated_last_utilization[i]) for
i in range(1, 9)}
204  mean_values_arrival_rate = {i: np.mean(aggregated_mean_arrival_rates[i])
for i in range(1, 9)}
205
206  for i in range(1, 9):
207      print(f'Mean value of the queue length for Stream {i} is',
mean_values_queue_length[i])
208      print(f'Mean value of the number of patients for Stream {i} is',
mean_values_num_patient[i])
209      print(f'Mean value of the wait time for Stream {i} is',
mean_values_wait_time[i])
210      print(f'Mean value of the nonzero wait time for Stream {i} is',
mean_values_nonzero_wait_time[i])
211      print(f'Mean value of the service time for Stream {i} is',
mean_values_service_time[i])
212      print(f'Mean value of the sojourn time for Stream {i} is',
mean_values_sojourn_time[i])
213      print(f'Mean value of the utilization for Stream {i} is',
mean_values_utilization[i])
214      print(f'Mean value of the arrival rate for Stream {i} is',
mean_values_arrival_rate[i])
215      print(f'Mean value of the throughput for Stream {i} is',
mean_values_throughput[i])
216      print("        ")
217      print(f'95% confidence interval of the queue length for Stream {i} is'
, confidence_intervals_queue_length[i])
218      print(f'95% confidence interval of the number of patients for Stream {
i} is', confidence_intervals_num_patient[i])
```

```
219         print(f'95% confidence interval of the wait time for Stream {i} is',
    confidence_intervals_wait_time[i])
220         print(f'95% confidence interval of the nonzero wait time for Stream {i
    } is', confidence_intervals_nonzero_wait_time[i])
221         print(f'95% confidence interval of the service time for Stream {i} is'
    , confidence_intervals_service_time[i])
222         print(f'95% confidence interval of the sojourn time for Stream {i} is'
    , confidence_intervals_sojourn_time[i])
223         print(f'95% confidence interval of the utilization for Stream {i} is',
     confidence_intervals_utilization[i])
224         print(f'95% confidence interval of the arrival rate for Stream {i} is'
    , confidence_intervals_arrival_rate[i])
225         print(f'95% confidence interval of the throughput for Stream {i} is',
    confidence_intervals_throughput[i])
226         print('-----------------------------------')
227
228 if __name__ == '__main__':
229     main()
```

Listing E.1: Statistical inference

APPENDIX F

PYTHON SCRIPT FOR GOODNESS-OF-FIT TESTING OF NONZERO WAIT TIME DISTRIBUTIONS

streams_goodness_of_fit_test_nonzero_wait_times.py:

```
import json
import os
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import kstest, gamma

# The all_data.json file for this analysis is generated from
# streams_nonhomo_warmup_run1.py (see Appendix C)

def load_data_from_json(filename='all_data.json'):
    with open(filename, 'r') as f:
        data = json.load(f)
    deserialized_data = {}
    for key, value in data.items():
        if key == 'warm_up_period':
            deserialized_data[key] = value
        else:
            deserialized_data[key] = {int(k): v for k, v in value.items()}
    return deserialized_data

def perform_gamma_ks_test(nonzero_wait_times):
    flattened_nonzero_wait_times = [time for sublist in nonzero_wait_times.
   values() for time in sublist]

    # Fit the gamma distribution to the data
    shape, loc, scale = gamma.fit(flattened_nonzero_wait_times, floc=0)
```

```
27
28      # Perform K-S test
29      d, p_value = kstest(flattened_nonzero_wait_times, 'gamma', args=(shape,
        loc, scale))
30
31      return d, p_value, shape, scale  # Return shape and scale for reporting
32
33
34  def plot_multiple_cdfs(stream_ids, all_nonzero_wait_times, all_shapes,
        all_scales, colors, output_file):
35      from scipy.stats import gamma
36      plt.figure(figsize=(12, 10))
37
38      for i, stream_id in enumerate(stream_ids, start=1):
39          nonzero_wait_times = all_nonzero_wait_times[stream_id]
40          shape = all_shapes[stream_id]
41          scale = all_scales[stream_id]
42
43          sorted_data = np.sort(nonzero_wait_times)
44          yvals = np.arange(len(sorted_data)) / float(len(sorted_data))
45
46          # Empirical CDF
47          plt.subplot(2, 2, i)
48          plt.step(sorted_data, yvals, label="Empirical CDF", color=colors[(
        stream_id - 1) % len(colors)])
49
50          # Theoretical Gamma CDF
51          plt.plot(sorted_data, gamma.cdf(sorted_data, a=shape, loc=0, scale=
        scale),
52                   label=f"Theoretical Gamma CDF", color='red', lw=2)
53
54          plt.title(f'CDF of Nonzero Wait Times (Stream {stream_id})')
```

```
        plt.xlabel('Nonzero Wait Time')
        plt.ylabel('CDF')
        plt.legend()
        plt.grid(True)

    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()

def plot_distribution(metric_data, title, xlabel, ylabel, colors, stream_ids, output_file):
    plt.figure(figsize=(12, 10))
    for stream_id, data in metric_data.items():
        if stream_id in stream_ids:
            if isinstance(data, list) and data:
                try:
                    mean_value = np.mean(data)
                    max_value = max(data)
                    print(f'Mean {ylabel} for Stream {stream_id}: {mean_value}')
                    print(f'Max {ylabel} for Stream {stream_id}: {max_value}')
                    plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
                    # Plot histogram and KDE with clipping at 0
                    sns.histplot(data, kde=False, color=colors[(stream_id - 1)], stat="density", alpha=0.6)
                    sns.kdeplot(data, color='black', lw=2, clip=(0, np.inf))  # Ensure no values below 0
                    plt.axvline(mean_value, color='r', linestyle='--', label=f'Mean = {mean_value:.2f}')
                    plt.xlabel(xlabel)
                    plt.ylabel(ylabel)
                    plt.title(f'{title} for Stream {stream_id}')
```

```
                    plt.legend()
                    plt.grid(True)
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list format.")
    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()

def main():
    # Create directory for plots
    output_dir = 'streams_nonzero_wait_times_gamma_test'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    all_data = load_data_from_json('all_data.json')

    # Store shape and scale parameters for each stream
    all_shapes = {}
    all_scales = {}

    # Perform Gamma K-S test on nonzero waiting time data for each stream
    for stream_id, nonzero_wait_times in all_data['nonzero_wait_times'].items():
        d, p_value, shape, scale = perform_gamma_ks_test({stream_id: nonzero_wait_times})
        all_shapes[stream_id] = shape
        all_scales[stream_id] = scale
        print(f'Stream {stream_id}: D = {d}, p-value = {p_value}, shape = {shape}, scale = {scale}')
```

```
110
111     # Define colors for streams
112     colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']
113
114     # Plot and save CDFs for streams 1-4 and 5-8, with different colors for
    each stream
115     plot_multiple_cdfs(range(1, 5), all_data['nonzero_wait_times'], all_shapes
    , all_scales,
116                        colors, os.path.join(output_dir, '
    cdf_nonzero_wait_times_1_4.png'))
117     plot_multiple_cdfs(range(5, 9), all_data['nonzero_wait_times'], all_shapes
    , all_scales,
118                        colors, os.path.join(output_dir, '
    cdf_nonzero_wait_times_5_8.png'))
119
120     # Plot and save nonzero waiting time distributions for streams 1-4 and 5-8
121     plot_distribution(all_data['nonzero_wait_times'], 'Nonzero Wait Time
    Distribution',
122                       'Nonzero Wait Time', 'Density', colors, range(1, 5),
123                       os.path.join(output_dir, '
    nonzero_wait_time_distribution_1_4.png'))
124     plot_distribution(all_data['nonzero_wait_times'], 'Nonzero Wait Time
    Distribution',
125                       'Nonzero Wait Time', 'Density', colors, range(5, 9),
126                       os.path.join(output_dir, '
    nonzero_wait_time_distribution_5_8.png'))
127
128 if __name__ == "__main__":
129     main()
```

Listing F.1: Goodness-of-Fit testing of nonzero wait time distributions

APPENDIX G

PYTHON SCRIPT FOR GOODNESS-OF-FIT TESTING OF SOJOURN TIME DISTRIBUTIONS

streams_goodness_of_fit_test_sojourn_times.py:

```
import json
import os
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import kstest, expon

# The all_data.json file for this analysis is generated from
# streams_nonhomo_warmup_run1.py (see Appendix C)

def load_data_from_json(filename='all_data.json'):
    with open(filename, 'r') as f:
        data = json.load(f)
    deserialized_data = {}
    for key, value in data.items():
        if key == 'warm_up_period':
            deserialized_data[key] = value
        else:
            deserialized_data[key] = {int(k): v for k, v in value.items()}
    return deserialized_data


def perform_exponential_ks_test(sojourn_times):
    flattened_sojourn_times = np.concatenate([np.array(sublist) for sublist in
    sojourn_times.values()])

    # Allow non-negative values, including zeros
```

```
    valid_sojourn_times = flattened_sojourn_times[flattened_sojourn_times >= 0]

    if len(valid_sojourn_times) == 0:
        raise ValueError("No valid data available for fitting the Exponential distribution.")

    # Fit the exponential distribution to the data
    loc, scale = expon.fit(valid_sojourn_times, floc=0)

    # Perform K-S test
    d, p_value = kstest(valid_sojourn_times, 'expon', args=(loc, scale))

    return d, p_value, loc, scale


def plot_multiple_cdfs(stream_ids, all_sojourn_times, all_scales, colors, output_file):
    from scipy.stats import expon
    plt.figure(figsize=(12, 10))

    for i, stream_id in enumerate(stream_ids, start=1):
        sojourn_times = all_sojourn_times[stream_id]
        scale = all_scales[stream_id]  # No shape parameter for Exponential distribution

        sorted_data = np.sort(sojourn_times)
        yvals = np.arange(len(sorted_data)) / float(len(sorted_data))

        # Empirical CDF
        plt.subplot(2, 2, i)
        plt.step(sorted_data, yvals, label="Empirical CDF", color=colors[(
```

```
    stream_id - 1) % len(colors)])

        # Theoretical Exponential CDF
        plt.plot(sorted_data, expon.cdf(sorted_data, loc=0, scale=scale),
                 label=f"Theoretical Exponential CDF", color='red', lw=2)

        plt.title(f'CDF of Sojourn Times (Stream {stream_id})')
        plt.xlabel('Sojourn Time')
        plt.ylabel('CDF')
        plt.legend()
        plt.grid(True)

    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()


def plot_distribution(metric_data, title, xlabel, ylabel, colors, stream_ids,
    output_file):
    plt.figure(figsize=(12, 10))
    for stream_id, data in metric_data.items():
        if stream_id in stream_ids:
            if isinstance(data, list) and data:
                try:
                    mean_value = np.mean(data)
                    max_value = max(data)
                    print(f'Mean {ylabel} for Stream {stream_id}: {mean_value}
    ')
                    print(f'Max {ylabel} for Stream {stream_id}: {max_value}')
                    plt.subplot(2, 2, (stream_id - 1) % 4 + 1)
                    # Plot histogram and KDE with clipping at 0
                    sns.histplot(data, kde=False, color=colors[(stream_id - 1)
```

```
    ], stat="density", alpha=0.6)
                    sns.kdeplot(data, color='black', lw=2, clip=(0, np.inf))
    # Ensure no values below 0
                    plt.axvline(mean_value, color='r', linestyle='--', label=f
    'Mean = {mean_value:.2f}')
                    plt.xlabel(xlabel)
                    plt.ylabel(ylabel)
                    plt.title(f'{title} for Stream {stream_id}')
                    plt.legend()
                    plt.grid(True)
                except ValueError as e:
                    print(f"Error processing stream {stream_id}: {e}")
            else:
                print(f"No data for Stream {stream_id} or data is not in list
    format.")
    plt.tight_layout()
    plt.savefig(output_file)
    plt.show()


def main():
    # Create directory for plots
    output_dir = 'streams_sojourn_times_expon_test'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    all_data = load_data_from_json('all_data.json')

    # Store scale parameters for each stream
    all_scales = {}

    # Perform Exponential K-S test on system time data for each stream
```

```
    for stream_id, sojourn_times in all_data['sojourn_times'].items():
        d, p_value, loc, scale = perform_exponential_ks_test({stream_id: sojourn_times})
        all_scales[stream_id] = scale
        print(f'Stream {stream_id}: D = {d}, p-value = {p_value}, scale = {scale}')

    # Define colors for streams
    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']

    # Plot and save CDFs for streams 1-4 and 5-8, with different colors for each stream
    plot_multiple_cdfs(range(1, 5), all_data['sojourn_times'], all_scales,
                       colors, os.path.join(output_dir, 'cdf_sojourn_times_1_4.png'))
    plot_multiple_cdfs(range(5, 9), all_data['sojourn_times'], all_scales,
                       colors, os.path.join(output_dir, 'cdf_sojourn_times_5_8.png'))

    # Plot and save system time distributions for streams 1-4 and 5-8
    plot_distribution(all_data['sojourn_times'], 'Sojourn Time Distribution',
                      'Sojourn Time', 'Density', colors, range(1, 5),
                      os.path.join(output_dir, 'sojourn_time_distribution_1_4.png'))
    plot_distribution(all_data['sojourn_times'], 'Sojourn Time Distribution',
                      'Sojourn Time', 'Density', colors, range(5, 9),
                      os.path.join(output_dir, 'sojourn_time_distribution_5_8.png'))

if __name__ == "__main__":
    main()
```

Listing G.1: Goodness-of-Fit testing of sojourn time distributions

APPENDIX H

PYTHON SCRIPT PLOTTING UTILIZATION AND THROUGHPUT OVER TIME

utilization_throughput.py:

```
import json
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# The all_data.json file for this analysis is generated from
# streams_nonhomo_warmup_run1.py (see Appendix C)

def load_data_from_json(filename):
    with open(filename, 'r') as f:
        data = json.load(f)
    return data

def calculate_statistics(data):
    df = pd.DataFrame(data)
    stats = df.describe()
    return stats

def plot_utilization_and_throughput(utilization_data, throughput_data, ids,
    colors, output_file, title_prefix):
    plt.figure(figsize=(12, 10))
    for idx, color in zip(ids, colors):
        util_times, util_values = zip(*utilization_data[str(idx)])
        thr_times, thr_values = zip(*throughput_data[str(idx)])

        plt.subplot(2, 2, (idx - 1) % 4 + 1)
        plt.plot(util_times, util_values, color=color, linestyle='-', label=f'
```

```
    Utilization {title_prefix} {idx}')
28          plt.plot(thr_times, thr_values, color=color, linestyle='--', label=f'
    Throughput {title_prefix} {idx}')
29          plt.xlabel('Time')
30          plt.ylabel('Value')
31          plt.title(f'Utilization and Throughput for {title_prefix} {idx}')
32          plt.legend()
33          plt.grid(True)
34
35      plt.tight_layout()
36      plt.savefig(output_file)
37      plt.show()
38
39  def main():
40      # Create directory for plots
41      output_dir = 'utilization_throughput_correlation'
42      if not os.path.exists(output_dir):
43          os.makedirs(output_dir)
44
45      # Analysis for streams from all_data.json
46      all_data = load_data_from_json('all_data.json')
47
48      # Extract data for analysis
49      utilization_data = {}
50      throughput_data = {}
51
52      for stream_id in range(1, 9):
53          util_times, util_values = zip(*all_data['utilization'][str(stream_id)
    ])
54          thr_times, thr_values = zip(*all_data['throughput'][str(stream_id)])
55
56          # Ensure arrays are of the same length by trimming to the minimum
```

```
length
57          min_length = min(len(util_values), len(thr_values))
58          utilization_data[stream_id] = util_values[:min_length]
59          throughput_data[stream_id] = thr_values[:min_length]
60
61          print(f"Stream {stream_id}: Utilization length = {len(util_values)},
    Throughput length = {len(thr_values)}")
62          print(f"Stream {stream_id}: Trimmed Utilization length = {len(
    utilization_data[stream_id])}, Trimmed Throughput length = {len(
    throughput_data[stream_id])}")
63          print(f"Stream {stream_id}: Final Utilization length = {len(
    utilization_data[stream_id])}, Final Throughput length = {len(
    throughput_data[stream_id])}")
64
65      # Convert to DataFrame for each stream separately
66      utilization_dfs = []
67      throughput_dfs = []
68
69      for stream_id in range(1, 9):
70          utilization_dfs.append(pd.DataFrame(utilization_data[stream_id],
    columns=[f'Stream {stream_id}']))
71          throughput_dfs.append(pd.DataFrame(throughput_data[stream_id], columns
    =[f'Stream {stream_id}']))
72
73      utilization_df = pd.concat(utilization_dfs, axis=1)
74      throughput_df = pd.concat(throughput_dfs, axis=1)
75
76      # Calculate statistics
77      utilization_stats = utilization_df.describe()
78      throughput_stats = throughput_df.describe()
79
80      print("Utilization Statistics:")
```

```
    print(utilization_stats)
    print("\nThroughput Statistics:")
    print(throughput_stats)

    # Plot utilization and throughput together for each stream
    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']  # Colors for 8 streams
    plot_utilization_and_throughput(all_data['utilization'], all_data['throughput'], range(1, 5), colors[:4], os.path.join(output_dir, 'utilization_and_throughput_1_4.png'), 'Stream')
    plot_utilization_and_throughput(all_data['utilization'], all_data['throughput'], range(5, 9), colors[4:], os.path.join(output_dir, 'utilization_and_throughput_5_8.png'), 'Stream')

    # Calculate correlation
    correlation = utilization_df.corrwith(throughput_df)

    print("\nCorrelation between Utilization and Throughput:")
    print(correlation)

if __name__ == "__main__":
    main()
```

Listing H.1: Utilization and throughput over time

APPENDIX I

PYTHON SCRIPT FOR UTILIZATION AND NONZERO WAIT TIMES

correlation_util_nonzero_wt.py:

```
import simpy
import numpy as np
import random
import itertools
import math
import json
import os
import matplotlib.pyplot as plt
import time

class PatientService:
    def __init__(self, env, num_beds, servicetime):
        self.env = env
        self.bed = simpy.Resource(env, num_beds)
        self.servicetime = servicetime

    def serve(self, patient):
        yield self.env.timeout(np.random.exponential(scale=self.servicetime))

def patient(env, name, ps, num_beds, proportion, stream_id, results):
    arrive = env.now
    if random.random() <= proportion:
        results["leaving_patients"][stream_id].append((arrive, stream_id))
        return

    if arrive > results["WARM_UP_PERIOD"]:
        results["queue_lengths"][stream_id].append((env.now, len(ps.bed.queue)
    ))
```

```
        results["num_patients"][stream_id].append((env.now, len(ps.bed.queue) + len(ps.bed.users)))

    with ps.bed.request() as request:
        yield request
        wait = env.now - arrive
        if arrive > results["WARM_UP_PERIOD"]:
            results["waiting_times"][stream_id].append(wait)
            if wait > 0:
                results["nonzero_waiting_times"][stream_id].append(wait)
        service_start = env.now
        yield env.process(ps.serve(name))
        service_time = env.now - service_start
        if arrive > results["WARM_UP_PERIOD"]:
            results["service_times"][stream_id].append(service_time)
            results["system_times"][stream_id].append(wait + service_time)
            results["total_service_time"][stream_id] += service_time
            utilization = results["total_service_time"][stream_id] / (num_beds * (env.now - results["WARM_UP_PERIOD"]))
            results["utilization_over_time"][stream_id].append((env.now, utilization))
            results["served"][stream_id] += 1
            results["throughput"][stream_id].append((env.now, results["served"][stream_id] / (env.now - results["WARM_UP_PERIOD"])))

def setup(env, base_arrival_rate, service_rate, num_beds, proportion, stream_id, results):
    patientservice = PatientService(env, num_beds, 1 / service_rate)
    patient_count = itertools.count()
    max_arrival_rate = base_arrival_rate * 1.1  # Maximum rate considering the sinusoidal fluctuation

```

```
    while True:
        interarrival_time = np.random.exponential(scale=1 / max_arrival_rate)
        yield env.timeout(interarrival_time)
        t = env.now

        current_arrival_rate = base_arrival_rate + 0.1 * base_arrival_rate *
    math.sin(math.pi * t / 12)

        if random.random() <= current_arrival_rate / max_arrival_rate:
            env.process(patient(env, f'Patient {next(patient_count)}',
    patientservice, num_beds, proportion, stream_id, results))

def run_simulation(run, params, whole_run_leaving_patients, WARM_UP_PERIOD
    =10000, SIM_TIME=30000):
    # RANDOM_SEED = random.randint(40, 50)
    # random.seed(RANDOM_SEED)
    results = {
        "queue_lengths": {i: [] for i in range(1, 9)},
        "num_patients": {i: [] for i in range(1, 9)},
        "system_times": {i: [] for i in range(1, 9)},
        "throughput": {i: [] for i in range(1, 9)},
        "served": {i: 0 for i in range(1, 9)},
        "leaving_patients": {i: [] for i in range(1, 9)},
        "waiting_times": {i: [] for i in range(1, 9)},
        "nonzero_waiting_times": {i: [] for i in range(1, 9)},
        "service_times": {i: [] for i in range(1, 9)},
        "utilization_over_time": {i: [] for i in range(1, 9)},
        "total_service_time": {i: 0 for i in range(1, 9)},
        "WARM_UP_PERIOD": WARM_UP_PERIOD
    }
    for param in params:
        env = simpy.Environment()
```

```
        env.process(setup(env, param["arrival_rate"], param["service_rate"], param["num_beds"], param["proportion"], param["stream_id"], results))
        env.run(until=SIM_TIME)

    mean_nonzero_waiting_times = {
        i: np.mean(results["nonzero_waiting_times"][i]) if results["nonzero_waiting_times"][i] else 0
        for i in range(1, 9)
    }
    last_utilization = {
        i: results["utilization_over_time"][i][-1][1] if results["utilization_over_time"][i] else 0
        for i in range(1, 9)
    }

    whole_run_leaving_patients[run] = results["leaving_patients"]

    return last_utilization, mean_nonzero_waiting_times

def main():
    # Check if the data file exists
    if os.path.exists('utilization_nonzero_waiting_time_data.json'):
        # Load the data if the file exists
        with open('utilization_nonzero_waiting_time_data.json', 'r') as f:
            data = json.load(f)
            all_mean_utilization = data['all_mean_utilization']
            all_mean_nonzero_waiting_time = data['all_mean_nonzero_waiting_time']
        print("Data loaded from 'utilization_nonzero_waiting_time_data.json'.")
    else:
        # Generate the data if the file does not exist
```

```
    original_params = [
        {"arrival_rate": 24.5 / 24, "service_rate": 1 / (2.37 * 24), "num_beds": 65, "proportion": 0.0, "stream_id": 1},
        {"arrival_rate": 37 / 24, "service_rate": 1 / (3.02 * 24), "num_beds": 119, "proportion": 0.0, "stream_id": 2},
        {"arrival_rate": 26 / 24, "service_rate": 1 / (4.09 * 24), "num_beds": 115, "proportion": 0.0, "stream_id": 3},
        {"arrival_rate": 19 / 24, "service_rate": 1 / (2.89 * 24), "num_beds": 62, "proportion": 0.0, "stream_id": 4},
        {"arrival_rate": 16.5 / 24, "service_rate": 1 / (3.27 * 24), "num_beds": 62, "proportion": 0.0, "stream_id": 5},
        {"arrival_rate": 14.1 / 24, "service_rate": 1 / (3.51 * 24), "num_beds": 57, "proportion": 0.0, "stream_id": 6},
        {"arrival_rate": 9.8 / 24, "service_rate": 1 / (5.56 * 24), "num_beds": 65, "proportion": 0.0, "stream_id": 7},
        {"arrival_rate": 16.3 / 24, "service_rate": 1 / (4.63 * 24), "num_beds": 85, "proportion": 0.0, "stream_id": 8}
    ]

    all_mean_utilization = {i: [] for i in range(1, 9)}
    all_mean_nonzero_waiting_time = {i: [] for i in range(1, 9)}

    for percentage in range(100, 70, -1):  # From 100% to 60%, decreasing by 2%
        modified_params = []
        for param in original_params:
            modified_param = param.copy()
            modified_param['arrival_rate'] *= (percentage / 100.0)
            modified_params.append(modified_param)

        # Run the simulation for this set of modified_params
        mean_values_utilization, mean_values_nonzero_waiting_time =
```

```
run_simulation_with_params(modified_params)

            # Store the results for correlation analysis
            for i in range(1, 9):
                all_mean_utilization[i].append(mean_values_utilization[i])
                all_mean_nonzero_waiting_time[i].append(
mean_values_nonzero_waiting_time[i])

        # Save the data for all_mean_utilization and
all_mean_nonzero_waiting_time into a JSON file
        data = {
            'all_mean_utilization': all_mean_utilization,
            'all_mean_nonzero_waiting_time': all_mean_nonzero_waiting_time
        }

        with open('utilization_nonzero_waiting_time_data.json', 'w') as f:
            json.dump(data, f, default=convert_to_serializable)

        print("Data generated and saved to '
utilization_nonzero_waiting_time_data.json'.")

    # Create a folder for the plots
    output_folder = 'scatter_plots'
    os.makedirs(output_folder, exist_ok=True)

    # Plot the scatter plots for each stream
    plot_scatter(all_mean_utilization, all_mean_nonzero_waiting_time,
output_folder)

    # Calculate and print correlation coefficients for each stream
    calculate_and_print_correlation(all_mean_utilization,
all_mean_nonzero_waiting_time)
```

```
159
160 # Assuming run_simulation_with_params is a function that takes modified_params
        as input and returns mean_values_utilization and
    mean_values_nonzero_waiting_time
161 def run_simulation_with_params(params):
162     num_runs = 30
163     warm_up_period = 10000
164     sim_time = 30000
165
166     aggregated_mean_utilization = {i: [] for i in range(1, 9)}
167     aggregated_mean_nonzero_waiting_time = {i: [] for i in range(1, 9)}
168
169     # Run the simulations (this may be parallelized as before)
170     for run in range(1, num_runs + 1):
171         # run_simulation should be adapted to take params as input directly
172         lu, mnwt = run_simulation(run, params, {}, warm_up_period, sim_time)
173
174         # Collect mean utilization and nonzero waiting times for this run
175         for i in range(1, 9):
176             if i in lu and lu[i]:
177                 aggregated_mean_utilization[i].append(lu[i])
178             if i in mnwt and mnwt[i]:
179                 aggregated_mean_nonzero_waiting_time[i].append(mnwt[i])
180
181     # Calculate the mean values across all runs
182     mean_values_utilization = {i: np.mean(aggregated_mean_utilization[i]) for
    i in range(1, 9)}
183     mean_values_nonzero_waiting_time = {
184         i: np.mean(aggregated_mean_nonzero_waiting_time[i]) if
    aggregated_mean_nonzero_waiting_time[i] else 0
185         for i in range(1, 9)
186     }
```

```
187
188     return mean_values_utilization, mean_values_nonzero_waiting_time
189
190 # Plot scatter plots
191 def plot_scatter(all_mean_utilization, all_mean_nonzero_waiting_time,
    output_folder):
192     colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange']
193
194     # Plot the first four streams in one figure
195     plt.figure(figsize=(12, 8))
196     for stream_id in range(1, 5):
197         plt.scatter(all_mean_utilization[str(stream_id)],
    all_mean_nonzero_waiting_time[str(stream_id)],
198                     color=colors[stream_id - 1], alpha=0.7, label=f'Stream {
    stream_id}')
199     plt.title('Streams 1-4: Mean Utilization vs Mean Nonzero Waiting Time')
200     plt.xlabel('Mean Utilization')
201     plt.ylabel('Mean Nonzero Waiting Time')
202     plt.grid(True)
203     plt.legend()
204     plt.savefig(os.path.join(output_folder, 'streams_1_to_4_scatter_plot.png')
    )
205     plt.show()
206
207     # Plot the last four streams in another figure
208     plt.figure(figsize=(12, 8))
209     for stream_id in range(5, 9):
210         plt.scatter(all_mean_utilization[str(stream_id)],
    all_mean_nonzero_waiting_time[str(stream_id)],
211                     color=colors[stream_id - 1], alpha=0.7, label=f'Stream {
    stream_id}')
212     plt.title('Streams 5-8: Mean Utilization vs Mean Nonzero Waiting Time')
```

```
    plt.xlabel('Mean Utilization')
    plt.ylabel('Mean Nonzero Waiting Time')
    plt.grid(True)
    plt.legend()
    plt.savefig(os.path.join(output_folder, 'streams_5_to_8_scatter_plot.png'))
    plt.show()

# Calculate and print correlation coefficients
def calculate_and_print_correlation(all_mean_utilization, all_mean_nonzero_waiting_time):
    for stream_id in range(1, 9):
        # Calculate the correlation coefficient
        correlation_coefficient = np.corrcoef(all_mean_utilization[str(stream_id)], all_mean_nonzero_waiting_time[str(stream_id)])[0, 1]
        print(f'Stream {stream_id}: Correlation Coefficient between Mean Utilization and Mean Nonzero Waiting Time: {correlation_coefficient:.4f}')

# When saving data to JSON, the script converts numpy arrays and numpy-specific types to Python lists and native types
def convert_to_serializable(data):
    if isinstance(data, np.ndarray):
        return data.tolist()
    elif isinstance(data, (np.float32, np.float64)):
        return float(data)
    elif isinstance(data, (np.int32, np.int64)):
        return int(data)
    return data

if __name__ == '__main__':
    main()
```

Listing I.1: Utilization and nonzero wait times

APPENDIX J

PYTHON SCRIPT FOR OPTIMAL BED ALLOCATION UNDER STATIC ARRIVAL RATES

sa_theory.py:

```
import numpy as np
import random
import math
from tqdm import tqdm

# Define the parameters for the hospital system
lambdas = np.array([24.5/24, 37/24, 26/24, 19/24, 16.5/24, 14.1/24, 9.8/24,
    16.3/24])
mus = np.array([1/(2.37*24), 1/(3.02*24), 1/(4.09*24), 1/(2.89*24),
    1/(3.27*24), 1/(3.51*24), 1/(5.56*24), 1/(4.63*24)])
total_servers = 630

# Function to calculate the waiting time for a given k
def calculate_waiting_time(k, lam, mu):
    rho = lam / (k * mu)
    if rho >= 1:
        return float('inf')  # Return infinity if the system is unstable
    return (1 / ((1 - rho) * k * mu))**2

# Objective function to minimize: the total waiting time across all streams
def total_waiting_time(k_values, lambdas, mus):
    total_w = 0
    for i in range(8):
        w_i = calculate_waiting_time(k_values[i], lambdas[i], mus[i])
        if w_i == float('inf'):
            return float('inf')
        total_w += w_i
```

```
    return total_w

# Simulated Annealing algorithm
def simulated_annealing(initial_state, lambdas, mus, total_servers, temp=100000, cooling_rate=0.995, max_iter=10000):
    current_state = initial_state.copy()
    current_energy = total_waiting_time(current_state, lambdas, mus)
    best_state = current_state.copy()
    best_energy = current_energy

    # Monitoring the initial state
    iteration = 0
    print(f"Iteration {iteration}, Temperature: {temp:.2f}, Current Energy: {current_energy}, Best Energy: {best_energy}, State: {current_state}")

    for i in tqdm(range(max_iter)):
        temp *= cooling_rate
        if temp <= 0.01:
            break

        # Dynamically adjust server change magnitude based on temperature
        change_magnitude = max(1, int(temp / 1000))  # Larger changes at higher temperatures
        new_state = current_state.copy()

        stream_a_idx = random.randint(0, 7)
        stream_b_idx = random.choice([i for i in range(8) if i != stream_a_idx])

        # Random adjustment with dynamic magnitude
        change = random.choice([-change_magnitude, change_magnitude])
        new_value_a = new_state[stream_a_idx] - change
```

```
        new_value_b = new_state[stream_b_idx] + change

        # Ensure values stay within bounds
        new_value_a = max(3, min(621, new_value_a))
        new_value_b = max(3, min(621, new_value_b))
        new_state[stream_a_idx], new_state[stream_b_idx] = new_value_a, new_value_b

        # Adjust total servers if off by 1 due to rounding (optional safeguard)
        if sum(new_state) != total_servers:
            adjustment_index = random.randint(0, 7)
            new_state[adjustment_index] += total_servers - sum(new_state)

        # Calculate new energy if total_servers constraint is met
        if sum(new_state) == total_servers:
            new_energy = total_waiting_time(new_state, lambdas, mus)
            delta_energy = new_energy - current_energy

            # Accept new state based on energy and temperature
            if delta_energy < 0 or random.uniform(0, 1) < math.exp(-delta_energy / temp):
                current_state, current_energy = new_state, new_energy

                # Update the best solution found
                if current_energy < best_energy:
                    best_state, best_energy = current_state.copy(), current_energy

        # Increment iteration count
        iteration += 1

```

```
        # Monitoring: print status every 1000 iterations
        if iteration % 1000 == 0:
            print(" ")
            print(f"Iteration {iteration}, Temperature: {temp:.2f}, Current Energy: {current_energy}, Best Energy: {best_energy}, State: {current_state}")

    if temp <= 0.01:
        print("Cooling process completed due to temperature reaching the minimum threshold.")
    else:
        print("Maximum iterations reached.")

    return best_state, best_energy

# Initialize the system with equal distribution of servers (can use any one of the following)
initial_k_values = np.array([65, 119, 115, 62, 62, 57, 65, 85])

# Ensure the total number of servers is correct in the initial state
assert sum(initial_k_values) == 630

# Parameters for the Simulated Annealing algorithm
initial_temp = 10000
cooling_rate = 0.995
max_iterations = 10000

# Run the Simulated Annealing algorithm
optimal_k_values, minimal_waiting_time = simulated_annealing(initial_k_values, lambdas, mus, total_servers, initial_temp, cooling_rate, max_iterations)

# Output the results
```

```
print(f"Optimal k values: {optimal_k_values}")
print(f"Minimal total waiting time: {minimal_waiting_time}")
```

Listing J.1: SA application with static arrival rates

APPENDIX K

PYTHON SCRIPT FOR OPTIMAL BED ALLOCATION UNDER DYNAMIC ARRIVAL RATES

sa_simu.py:

```
import numpy as np
import random
import math
import simpy
import multiprocessing as mp
import time
from tqdm import tqdm

lambdas = np.array([24.5 / 24, 37 / 24, 26 / 24, 19 / 24, 16.5 / 24, 14.1 /
    24, 9.8 / 24, 16.3 / 24])
mus = np.array([1 / (2.37 * 24), 1 / (3.02 * 24), 1 / (4.09 * 24), 1 / (2.89 *
     24), 1 / (3.27 * 24), 1 / (3.51 * 24), 1 / (5.56 * 24), 1 / (4.63 * 24)])
total_servers = 630

class PatientService:
    def __init__(self, env, num_beds, servicetime):
        self.env = env
        self.bed = simpy.Resource(env, num_beds)
        self.servicetime = servicetime

    def serve(self):
        yield self.env.timeout(np.random.exponential(scale=self.servicetime))

def patient(env, ps, stream_id, results):
    arrive = env.now
    with ps.bed.request() as request:
        yield request
```

```
        wait = env.now - arrive
        if wait > 0:
            results["nonzero_waiting_times"][stream_id].append(wait)
        yield env.process(ps.serve())

# for dynamic arrival rates simulation
def setup(env, base_arrival_rate, service_rate, num_beds, stream_id, results):
    patientservice = PatientService(env, num_beds, 1 / service_rate)
    max_arrival_rate = base_arrival_rate * 1.1  # Maximum rate considering the
    sinusoidal fluctuation

    while True:
        interarrival_time = np.random.exponential(scale=1 / max_arrival_rate)
        yield env.timeout(interarrival_time)
        t = env.now

        current_arrival_rate = base_arrival_rate + 0.1 * base_arrival_rate *
    math.sin(math.pi * t / 12)
        if random.random() <= current_arrival_rate / max_arrival_rate:
            env.process(patient(env, patientservice, stream_id, results))


def run_simulation(run, k_values, lambdas, mus, WARM_UP_PERIOD=10000, SIM_TIME
    =30000):
    results = {
        "nonzero_waiting_times": {i: [] for i in range(1, 9)},
    }
    for i in range(8):
        env = simpy.Environment()
        env.process(setup(env, lambdas[i], mus[i], k_values[i], i + 1, results
    ))
        env.run(until=WARM_UP_PERIOD)  # Run the warm-up period
```

```
54          results["nonzero_waiting_times"][i + 1].clear()  # Clear warm-up data
55          env.run(until=SIM_TIME)
56      mean_nonzero_waiting_times = {i: np.mean(results["nonzero_waiting_times"][
        i]) for i in range(1, 9) if results["nonzero_waiting_times"][i]}
57      return mean_nonzero_waiting_times
58
59
60  def total_waiting_time(k_values, lambdas, mus, num_runs=30, WARM_UP_PERIOD
        =10000, SIM_TIME=30000):   # each stream uses the mean of 30 replications
61      aggregated_mean_nonzero_waiting_times = {i: [] for i in range(1, 9)}
62      pool = mp.Pool(mp.cpu_count())
63
64      # Submit tasks asynchronously
65      async_results = [pool.apply_async(run_simulation, (run, k_values, lambdas,
         mus, 10000, 30000)) for run in range(1, num_runs + 1)]
66
67      # Close the pool and wait for the tasks to complete
68      pool.close()
69      pool.join()
70
71      # Collect the results
72      results = [res.get() for res in async_results]
73
74      for run, mnwt in enumerate(results, 1):
75          for i in range(1, 9):
76              if i in mnwt and mnwt[i]:
77                  aggregated_mean_nonzero_waiting_times[i].append(mnwt[i])
78
79      mean_values_nonzero_waiting_time = {i: np.mean(
        aggregated_mean_nonzero_waiting_times[i]) if
        aggregated_mean_nonzero_waiting_times[i] else 0 for i in range(1, 9)}
80
```

```
    # Ensure the values are valid numbers (e.g., replace NaNs)
    mean_values_nonzero_waiting_time = {i: (mean_values_nonzero_waiting_time[i] if not np.isnan(mean_values_nonzero_waiting_time[i]) else 0) for i in range(1, 9)}

    return np.sum([mean_values_nonzero_waiting_time[i] ** 2 for i in range(1, 9)])

# Simulated Annealing algorithm
def simulated_annealing(initial_state, lambdas, mus, total_servers, temp=100000, cooling_rate=0.995, max_iter=10000):
    current_state = initial_state.copy()
    current_energy = total_waiting_time(current_state, lambdas, mus, num_runs=30, WARM_UP_PERIOD=5000, SIM_TIME=10000)
    best_state = current_state.copy()
    best_energy = current_energy

    # Monitoring the loop status
    iteration = 0
    temperature_reached = False
    # print(f"Iteration {iteration}, Temperature: {temp:.2f}, Current Energy: {current_energy}, Best Energy: {best_energy}")
    print(f"Iteration {iteration}, Temperature: {temp:.2f}, Current Energy: {current_energy}, Best Energy: {best_energy}, State: {current_state}")

    for i in tqdm(range(max_iter)):
        temp *= cooling_rate
        if temp <= 0.01:
            temperature_reached = True
            break

        # Generate a new state by randomly adjusting the servers between
```

```
    streams
106         new_state = current_state.copy()
107         stream_a_idx = np.random.randint(0, 8)
108         stream_b_idx = np.random.choice([i for i in range(8) if i !=
    stream_a_idx])
109         if stream_a_idx != stream_b_idx:
110             change = random.choice([-2, -1, 1, 2])
111             new_value_a = new_state[stream_a_idx] - change
112             new_value_b = new_state[stream_b_idx] + change
113 
114             # Bounds check and correction
115             if new_value_a < 3:
116                 new_value_a = 3
117             if new_value_b < 3:
118                 new_value_b = 3
119             if new_value_a > 621:
120                 new_value_a = 621
121             if new_value_b > 621:
122                 new_value_b = 621
123 
124             new_state[stream_a_idx] = new_value_a
125             new_state[stream_b_idx] = new_value_b
126 
127             # Check to ensure the total number of servers is still correct
128             if sum(new_state) == total_servers:
129                 new_energy = total_waiting_time(new_state, lambdas, mus,
    num_runs=10, WARM_UP_PERIOD=10000, SIM_TIME=30000)
130 
131                 # Calculate acceptance probability
132                 delta_energy = new_energy - current_energy
133                 if delta_energy < 0 or np.random.uniform(0, 1) < math.exp(-
    delta_energy / temp):
```

```
                    current_state = new_state
                    current_energy = new_energy

                    # Update the best solution found
                    if current_energy < best_energy:
                        best_state = current_state.copy()
                        best_energy = current_energy

        # Increment iteration count
        iteration += 1

        # Monitoring: print status every 1000 iterations
        if iteration % 1000 == 0:
            print(f"Iteration {iteration}, Temperature: {temp:.2f}, Current Energy: {current_energy}, Best Energy: {best_energy}, State: {current_state}")

    if temperature_reached:
        print("Cooling process completed due to temperature reaching the minimum threshold.")
    else:
        print("Maximum iterations reached.")

    return best_state, best_energy

if __name__ == "__main__":
    initial_k_values = np.array([64, 119, 115, 62, 62, 58, 65, 85])   # static optimal solution

    # Ensure the total number of servers is 630 in the initial state
    assert sum(initial_k_values) == 630

```

```
    # Parameters for the Simulated Annealing algorithm
    initial_temp = 10000
    cooling_rate = 0.995
    max_iterations = 10000

    # Run the Simulated Annealing algorithm
    optimal_k_values, minimal_waiting_time = simulated_annealing(
initial_k_values, lambdas, mus, total_servers, initial_temp, cooling_rate,
     max_iterations)

    # Output the results
    print(f"Optimal k values: {optimal_k_values}")
    print(f"Minimal total waiting time: {minimal_waiting_time}")
```

Listing K.1: SA application with dynamic arrival rates

BIBLIOGRAPHY

[1] A. J. Singer, H. C. Jr. Thode, P. Viccellio, and J. M. Pines. The association between length of emergency department boarding and mortality. *Academic Emergency Medicine*, 18(12):1324–1329, 2011.

[2] Simon Jones, Chris Moulton, Simon Swift, Paul Molyneux, Steve Black, Neil Mason, Richard Oakley, and Clifford Mann. Association between delays to patient admission from the emergency department and all-cause 30-day mortality. *Emergency Medicine Journal*, 39(3):168–173, 2022.

[3] Nathan R Hoot and Dominik Aronsky. Systematic review of emergency department crowding: causes, effects, and solutions. *Annals of Emergency Medicine*, 52(2):126–136, 2008.

[4] Jesse M Pines, John A Hilton, Ellen J Weber, Alexander J Alkemade, Hazem Al Shabanah, Paul D Anderson, Michael Bernhard, Alessandro Bertini, Andreas Gries, Salvador Ferrandiz, Vimal A Kumar, Veli-Pekka Harjola, Brian Hogan, Bo Madsen, Suzanne Mason, Göran Ohlén, Timothy Rainer, Nicholas Rathlev, Elian Revue, Drew Richardson, and Michael J Schull. International perspectives on emergency department crowding. *Academic Emergency Medicine*, 18(12):1358–1370, 2011.

[5] Caroline Berchet. Emergency care services. (83), 2015.

[6] Hossein Reza Rasouli, A Aliakbar Esfahani, and M Abbasi Farajzadeh. Challenges, consequences, and lessons for way-outs to emergencies at hospitals: a systematic review study. *BMC Emergency Medicine*, 19(1):62, 2019.

[7] Mor Armony, Shlomo Israelit, Avishai Mandelbaum, Yariv N Marmor, Yulia Tseytlin, and Galit B Yom-Tov. On patient flow in hospitals: A data-based queueing-science perspective. *Stochastic Systems*, 5(1):146–194, 2015.

[8] Randolph Hall. *Queueing Methods—For Services and Manufacturing*. 10 1990.

[9] Randolph Hall, David Belson, Pavan Murali, and Maged Dessouky. *Modeling Patient Flows Through the Healthcare System*, pages 1–44. Springer US, Boston, MA, 2006.

[10] Ali Al Owad, Md Islam, Premaratne Samaranayake, and Azharul Karim. Relationships between patient flow problems, health care services, and patient satisfaction: an empirical investigation of the emergency department. *Business Process Management Journal*, 28, 05 2022.

[11] M. Ba-Aoum, N. Hosseinichimeh, K.P. Triantis, K. Pasupathy, M. Sir, and D. Nestler. Statistical analysis of factors influencing patient length of stay in emergency departments. *International Journal of Industrial Engineering and Operations Management*, 5(3):220–239, 2023.

[12] Hajnal Vass and Zsuzsanna Katalin Szabo. Application of queuing model to patient flow in emergency department. case study. *Procedia. Economics and finance*, 32:479–487, 2015.

[13] Nidal Hamza, Mazlina Majid, and Fadhl Hujainah. Sim-pfed: A simulation-based decision making model of patient flow for improving patient throughput time in emergency department. *IEEE Access*, PP:1–1, 07 2021.

[14] Matthew D McHugh, Linda H Aiken, Douglas M Sloane, Cathy Windsor, Christine Douglas, and Patsy Yates. Effects of nurse-to-patient ratio legislation on nurse staffing and patient mortality, readmissions, and length of stay: a prospective study in a panel of hospitals. *Lancet*, 397(10288):1905–1913, 2021.

[15] Karen B Lasater, Linda H Aiken, Douglas M Sloane, Rachel French, Brendan Martin, Kyrani Reneau, Maryann Alexander, and Matthew D McHugh. Chronic hospital nurse understaffing meets covid-19: an observational study. *BMJ Quality & Safety*, 30(8):639–647, 2021.

[16] S Sülz, A Fügener, M Becker-Peth, et al. The potential of patient-based nurse staffing: a queuing theory application in the neonatal intensive care setting. *Health Care Management Science*, 27:239–253, 2024.

[17] P Griffiths, C Saville, JE Ball, J Jones, T Monks, and Safer Nursing Care Tool study team. Beyond ratios - flexible and resilient nurse staffing options to deliver cost-effective hospital care and address staff shortages: A simulation and economic modelling study. *International Journal of Nursing Studies*, 117:103901, 2021.

[18] AJ Thomas Schneider, P Luuk Besselink, Maartje E Zonderland, Richard J Boucherie, Wilbert B van den Hout, Job Kievit, Paul Bilars, A Jaap Fogteloo, and Ton J Rabelink. Allocating emergency beds improves the emergency admission flow. *Interfaces*, 48(4):384–394, 2018.

[19] E Redondo, V Nicoletta, V Bélanger, JP Garcia-Sabater, P Landa, J Maheut, JA Marin-Garcia, and A Ruiz. A simulation model for predicting hospital occupancy for covid-19 using archetype analysis. *Healthcare Analytics*, 3:100197, 2023.

[20] Esra Bas. *Basics of probability and stochastic processes*. Springer, 2019.

[21] Luiz Pinto, Francisco Campos, Ignez Perpetuo, and Y.C.N.M.B. Ribeiro. Analisys of hospital bed capacity via queuing theory and simulation. *Proceedings - Winter Simulation Conference*, 2015:1281–1292, 01 2015.

[22] Pengyi Shi, Mabel C Chou, Jim G Dai, Ding Ding, and Joe Sim. Models and insights for hospital inpatient operations: Time-dependent ed boarding time. *Management Science*, 62(1):1–28, 2016.

[23] Zhongheng Zhang, Xiaodian Zhang, Shenhong Gu, Xiaoqing Xu, Wei Jiang, Chuanzhu Lv, and Shaojiang Zheng. Dynamic programming for solving a simulated clinical scenario of sepsis resuscitation. *Annals of Palliative Medicine*, 10(4), 2021.

[24] Arkaitz Artetxe, Manuel Graña, Andoni Beristain, and Sebastián Ríos. Balanced training of a hybrid ensemble method for imbalanced datasets: a case of emergency department readmission prediction. *Neural Computing and Applications*, 32, 05 2020.

[25] R El-Bouri, T Taylor, A Youssef, T Zhu, and DA Clifton. Machine learning in patient flow: a review. *Progress in Biomedical Engineering*, 3(2):022002, 2021.

[26] CH Smeltzer and L Curtis. An analysis of emergency department time: laying the groundwork for efficiency standards. *QRB. Quality Review Bulletin*, 13(7):240–242, 1987.

[27] J. William Schmidt and Robert Edward Taylor. *Simulation and Analysis of Industrial Systems*. R. D. Irwin, Homewood, Ill., 1970.

[28] Wikipedia contributors. Discrete-event simulation — Wikipedia, The Free Encyclopedia, 2024. [Online; accessed 27-September-2024].

[29] Kalasim contributors. Theory of Discrete-Event Simulation — Kalasim, 2024. [Online; accessed 27-September-2024].

[30] Averill M Law. Simulation modeling and analysis. fifth, 2015.

[31] Jerry Banks. *Discrete event system simulation*. Pearson Education India, 2005.

[32] Wikipedia contributors. Simulated annealing — Wikipedia, The Free Encyclopedia, 2024. [Online; accessed 26-July-2024].

[33] Nathan Rooy. Simulated annealing with python, 2020. [Online; accessed 26-July-2024].

[34] R.B. Cooper. *Introduction to Queueing Theory*. North Holland, 1981.