Article

Modeling Demand for Satellite Calls in a Remote Village

Chiara Piccolroaz [1], Dominik Childs [2]

[1] Hochschule München - University of Applied Sciences; dchilds@hm.edu

Version January 29, 2025 Submitted to Modeling and Simulation Assignment

Keywords: queue theory, modeling, simulation, optimization

Queuing has become a major component of many areas of our society, including services, traffic, networks, transport and telecommunications. In this study we consider a call shop on a remote island. The aim is to model its system and simulate how customers are served under different circumstances, including different arrival times resulting from seasonal peaks. The data used in this study are generated by an algorithm that simulates a negative exponential distribution of customer inter-arrival and service times, whose correctness is verified by the Kolmogorov-Smirnov test. The system implementation is based on an event-driven approach of an M/M/1 queueing model, whose correctness is proven by the Little's Law Theorem and a custom test. To better communicate the results, various statistics are computed and visualised. The results of these tests confirm the accuracy of the simulation. The simulation results show, that the call shop is capable of handling seasonal traffic peaks with a single satellite link, without continously making the system unstable.

1. Introduction

Queueing theory is the mathematical analysis of waiting lines. The core of queueing theory is the understanding of the progression of elements within a framework that provides insights into elements' flow movements under different circumstances, enables the prediction of systems' behaviour and performances, and facilitates the efficient allocation of resources necessary to provide services.

Its wide-ranging applications permeate many sectors, including telecommunications, computer networks, transport infrastructure and factory operations. For exle, in supermarkets, queueing theory can help decide how many checkout lanes to open to keep queues short, based on how quickly new customers are arriving and are being served. Similarly, in the digital world, queueing theory can help keep websites running smoothly by calculating how to balance server load and computing power so that everyone can access the site quickly and reliably, even at busy times.

In the case of this study, we focus on modelling a call shop that is located on a remote island and accommodates international calls over a single satellite link. The aim of this paper is to develop a tailored model of this call shop and to simulate how its customers are served. The primary objective is to predict the main metrics of this system's behaviour, such as mean queue and system length, mean queue and system time, and mean server load 3.3.1. The secondary objective is to assess whether the store can meet peak season demand with one satellite, or whether a second satellite should be purchased 3.3.2.

With an event-driven based implementation of an M/M/1 queueing model, this study contributes to correctly simulate the call shop and help the owner gain an overview of the shop's operation under different arrival times, service times, system loads and simulation durations.

2. Data and Methods

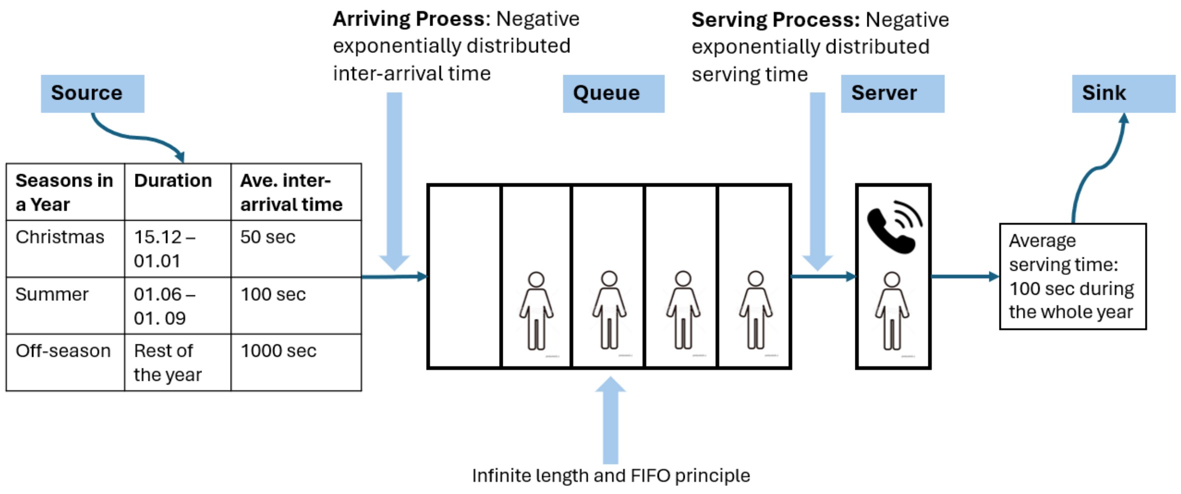
In order to enhance the understanding and replicability of this study, this section first presents the main characteristics and assumptions about the call shop 2.1, and then describes and motivates the data 2.2 and methods 2.3 used to model it.

2.1. Characteristics and Assumptions of the Call Shop

A queue is an ordered collection of entities or events that share one or more resources. Similarly, the system in the considered call shop operates as a queue, accommodating multiple customers waiting to use a common satellite for international calls. For a visual representation see figure 1.

Based on the given specifications, we assume that there is only one queue where customers can be ordered and one server where they can be served to make their call. To avoid complicating the model, for exle to prevent overflow, we have also chosen to treat the queue as an infinite line and to make the server work uninterrupted. According to the information provided, the queue works on a first-in-first-out (FIFO) basis, where new customers are added to the end of the queue and removed from the head.

In this setup, customers are said to arrive at independent intervals dictated by seasonal averages, which are 50 seconds at Christmas, 100 seconds in the summer and 1000 seconds otherwise. Once in the shop, customers either join a queue if the server is busy, or go directly to the server if it is free. They leave the shop after being served by the server, with each service taking an average of 100 seconds, regardless of the time of year.


Arriving Proess: Negative exponentially distributed inter-arrival time
Serving Process: Negative exponentially distributed serving time
Source
Queue
Server
Sink
Average serving time: 100 sec during the whole year
Infinite length and FIFO principle

Seasons in a Year	**Duration**	**Ave. inter-arrival time**
Christmas	15.12 – 01.01	50 sec
Summer	01.06 – 01. 09	100 sec
Off-season	Rest of the year	1000 sec

Figure 1. Characteristics and assumptions of the Call Shop

2.2. Data for Simulating the Call Shop

Besides the average time between arrival and service, there is no concrete data on the time customers arrive at the shop or spend at the server. Since these arrival and service times are stochastic processes essential for simulating the queue, we have to assume that they follow a certain distribution. Here we chose to assume a negative exponential distribution. The method and motivation for generating inter-arrival and service data that follow a negative exponential distribution is described below.

As shown in Listing 1, the data generation is performed by the method *drawNegativeExponentialDistributedNumber*, which uses the java.util.Random [1]. It takes the average arrival or service rate (λ) and a random number between 0 and 1 (μ) as variables and returns a random negative exponential value. This method is based on inverse transform sling, a basic technique for generating random numbers from any probability distribution given its cumulative distribution function [2]. In our case, $-\lambda \cdot \log(\mu)$ represents the inverse function of the cumulative distribution function of a negative exponential distribution.

In queueing theory, the exponential distribution is one of the most important probability distributions used to model the arrival and service time of each customer [3, p. 213-215]. A major advantage is that this distribution only requires the arrival and service rates λ to be computed. A further key feature of this distribution is the Markov property, which states that the probability of an event occurring in the next time interval is independent of the time already elapsed. Since in the call shop the arrival and service rates are the only available time data and are assumed to be independent of each other, the negative exponential distribution appears to be a very appropriate choice for this study.

```java
import java.util.Random;

public class Rng extends Random implements IRng {

    public Rng() {
        super();
    }

    public Rng(long seed) {
        super(seed);
    }

    public double drawNegativExponentialDistributedNumber(double lambda) {
    double u = nextDouble();
    return -lambda * Math.log(u);
    }
```

Listing 1: Generating negative exponential distributed data using inverse transform sampling

2.3. Methods for Simulating the Call Shop

In our study, we utilised the programming languages Java and Python, each for different components of our project.

Java was used to simulate the call shop. The main reason lies in the robust and object-oriented characteristics of this language, which allowed us to create seven classes that implement the custom interfaces needed to model the shop system. In addition, Java offers considerable performance standards, making it an appropriate choice for our simulation. The Java code was supplemented by the inclusion of specific packages such as lombok and java.util. The former automatically generated getters, setters and constructors, improving code clarity and reducing potential errors. The latter provided utilities such as LinkedList, Queue and Random, which are essential for data management in Java and are widely used in our code.

Python was used to print statistics, plot data and run tests on the data generated by the simulation. The main reason for this decision lies in the great interpretability and adaptability that this language offers. In fact, the seamless integration of Python packages such as numpy, pandas, seaborn and matplotlib allowed us to easily manipulate data, perform statistical analysis and create visual representations. These packages were complemented by the use of other standard libraries such as os, pathlib and scipy to meet our analytical needs.

We aimed to code according to the FAIR principles. Therefore, we provided a GitLab repository link to ensure that our code could be easily found https://gitlab.lrz.de/hm-koester-lectures/ss2024-modsim/assignment-2-callshop/-/tree/main/CiDo?ref_type=heads. We organised the codebase into a logical structure with detailed comments and documentation to make it easier for others to access and understand our work. Additionally, we tried to designed our code to be highly modular, generalisable and parameterisable to make it easier to adapt or reuse.

3. Results

This chapter is divided into four main sections covering the central aspects of our model, namely its theoretical background 3.1, technical implementation 3.2, results 3.3 and correctness 3.4.

3.1. Modelling the call shop

The main characteristics and assumptions of the call shop have been described in the previous chapter 2.1. In this section we place this information in the theoretical framework and present the approach chosen to implement the call-shop model.

In queueing theory, Kendall's notation is the standard system used to describe and classify a queueing system. The basic queueing models are described by the three factors A/S/c, where A is the time between arrivals in the queue, S is the service time distribution, and c is the number of service channels open at the node. In an M/M/1 queue system, the first and second M indicate that the time between arrivals and service is Markovian, i.e. the inter-arrival and service times follow an exponential distribution based on an average inter-arrival and service time, while the 1 indicates that there is only one service channel. This is also called a *single-server queueing system*.

Since the call shop under analysis exhibits exactly the same characteristics, we define it as an M/M/1 queueing system.

There are two main approaches to modelling queues in a system. The unit-time approach focuses on time, dividing it into equal discrete units. At each time step, the state of the system is updated whether or not particular events have occurred. The event-driven approach focuses on the occurrence of events. Instead of progressing in fixed time steps, the simulation jumps from one event to the next.

We chose the event-driven approach to simulate the call shop because it is typically more efficient for queueing models, as it does not waste computational resources on periods when there is no activity. In addition, it can handle complex interactions and variable processing times more naturally than the unit-time approach, since it only has to deal with changes in the system as they actually occur. However, it can be more complex to program because the logic must handle event dependencies and the order of events with precision. This often takes the form of an event schedule that keep track of upcoming events and time progressions.

3.2. Implementing the Model

In the following section, we delve into the technical and logical implementation of the call shop model.

As already mentioned in section 2.3 and shown in Listing 2, our project is divided into two main parts, respectively programmed in Java to simulate (*Simulation.java*) and test (*SimulationTest.Java*) the Call Shop model, and in Python to print statistics (*Statistics.py*), plot data (*Plots.py*) and perform tests (*KolmogorovSmirnovTest.py*) on the output generated by the simulation, which are stored as csv files in the *resources* folder. The Java implementation follows the rule "code against interfaces, not implementations", which means that the necessary functions are planned universally and specific ones are added during implementation.

```
- java/src/eventdrivensimulation/sim
    - Enums
        > Season.java
        > SeasonMultiplicationFactor.java
        > TimeUnits.java
    - interfaces
         > IClock.java
         > IQueue.java
         > IRng.java
```

```
          > ISimulation.java
          > IStatisticsWriter.java
    > Cleint.java
    > Clock.java
    > Queue.java
    > Rng.java
    > Server.java
    > Simulation.java
    > SimulationTest.java
    > StatisticsWriter.java
- python
    - plots
          > plots.py
    - resources
    - statisticalTests
        > KolmogorovSmirnovTest.py
        > Statistics.py
```

Listing 2: Structure of the Project

Of particular importance are the scripts *Cleint.java*, *Clock.java*, *Queue.java*, *Rng.java*, *Server.java*, which represent the main components of the Call Shop model (Table 1), and *Simulation.java*, which starts the simulation for specified parameters such as inter-arrival times, service times, duration of the simulation, seasonality.

Class	Use	Attributes
Client	This class handles specific information of the clients of the call shop	personId, interArrivalTime, arrivalTime, servingTime, queuingTime, storageTime
Clock	This class holds the arrival of the last client and the current time in the simulation	jump, time
Queue	This class contains information and functions related to the queue's length	queueLength
Rng	This class contains multiple constructors and functions to sample random numbers from different distributions. The most important is the *drawNegativeExponentialDistributedNumber* method described in section 2.2	
Server	This class tracks the end time of the current serving process as well as its length	endOfServing, serverLength

Table 1. Table of the classes representing the main components of the call shop

Figure 2 summarises the logical structure of the main *start*() method in *Simulation.java*, which is responsible for simulating the customer flow within the call shop model. Immediately after the function is called, it enters a loop that stops when the specified simulation duration is reached. In each loop, a client is processed depending on server availability and queue state, resulting in three possible scenarios:

1. The cenario called *case_PushNew* where the server is busy. In this case a newly created client is queued and the variable *createClient* is set to true so that a new client can be created in the next loop.

2. The cenario called *case_DirSer* where the server is free and there are no other clients in the queue. In this case the new client will bypass the queue and be served directly. The *createClient* variable will also be set to true.

3. The cenario called *case_PopFirst* where the server is free and the queue is empty. More specifically, the server was available between the arrival time of the last client in the queue and the newly created client. Since at least one client should have been served in this interval, the first client in the queue is removed and passed to the server, while the new client is not yet allowed to join the queue. In fact, the *createClient* flag is set to false, indicating that the current client should be reconsidered

in the next iteration, where is will wither passed to a further iteration via *case_PopFirst* or it will be served via *case_PushLast* or *case_DirSer* scenarios.

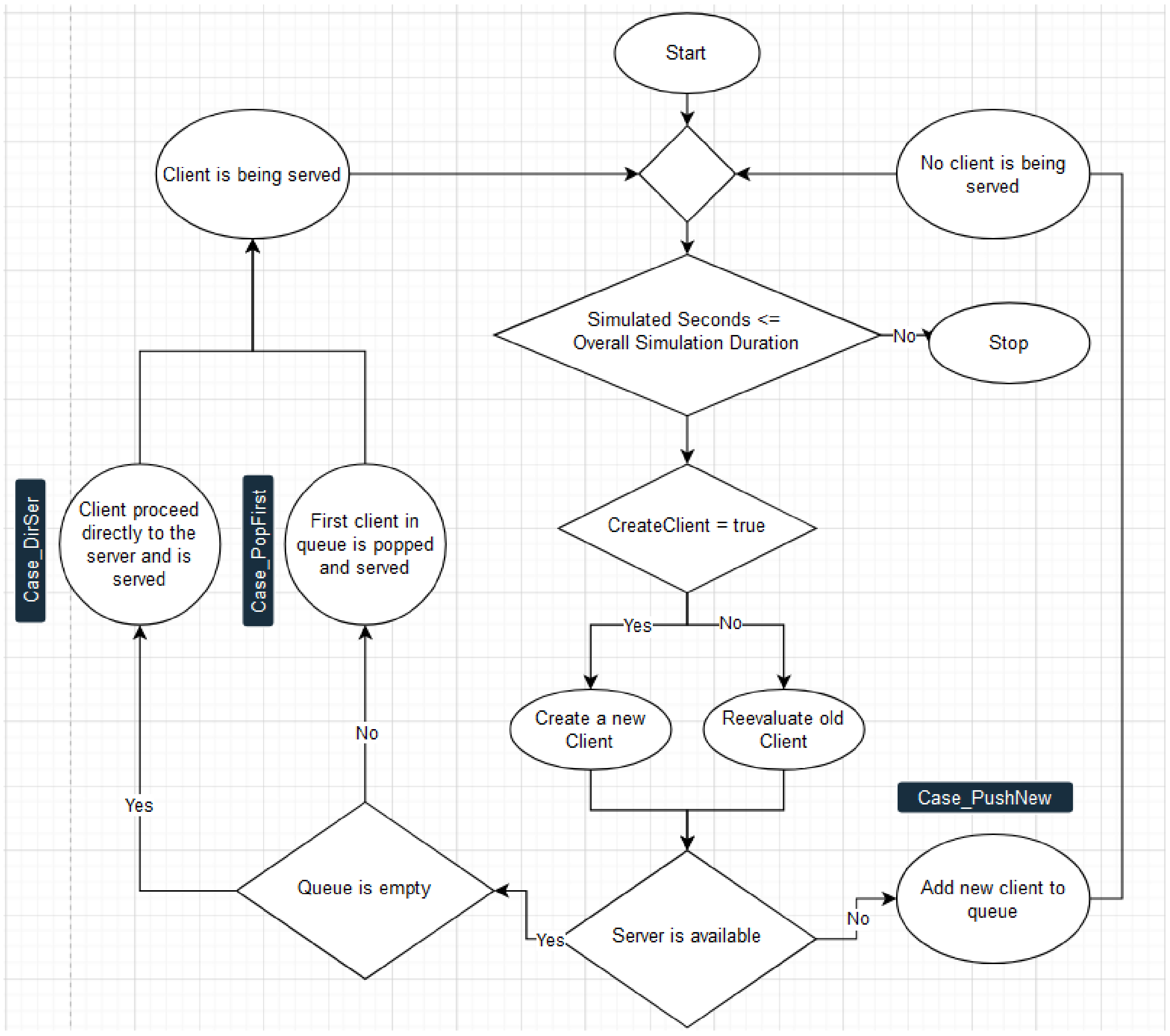


Figure 2. Flowchart of the model's simulation basic loop without accouting for peak seasons

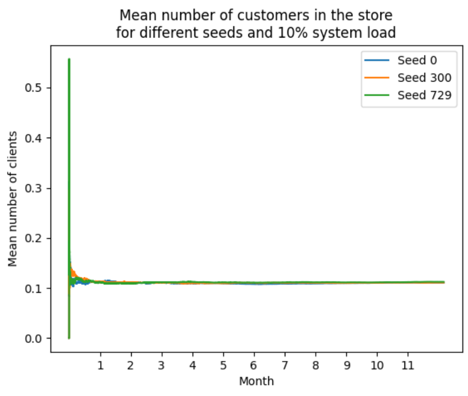
3.3. Simulation Results

In this section, we present the results of simulations designed to address the two main objectives of our study: predicting the main metrics of this system's behaviour and assessing whether the store can meet peak season demand.

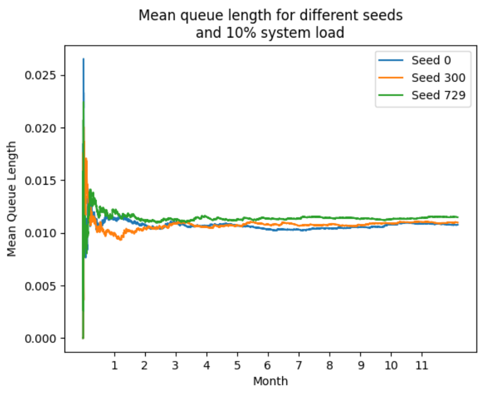
3.3.1. Constant Arrival Times

This section focuses on key metrics such as average queue and system length, average queue and system time, and average server load for constant arrival times. The graphs below show different metrics along the vertical axis while the passage of simulation time is displayed along the horizontal axis. As mentioned in the 2.1 section, there are two independent entities that drive the simulation: the clients and the server. More specifically, there are three events - when clients arrive, when service begins, and when service ends - at which the status of these two entities changes and therefore the metrics are updated. In the periods between these moments, metrics like queue and system length, time, and server load remain constant and are therefore not recorded in our simulation outputs. TThis approach corresponds to the event-driven method we use to simulate our M/M/1 queue model described in section 3.1. These time jumps are evenly distributed on the x-axis to give a linear visualisation of the simulation time.

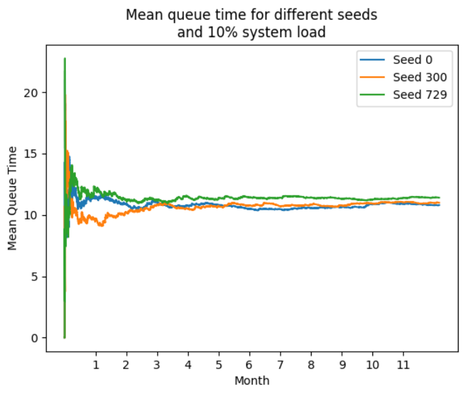
Figure 5 shows the one-year evolution of these metrics for simulations with off-season arrival times of 1000s and service times of 100s. The coloured lines in the plots represent different seeds for the negative exponential time generation, which produce consistent results across all metrics and simulations despite different data inputs. Overall, we can see that after one year, the average number of customers in the queue is 0.010, the average number of customers in the system is 0.11, the average time spent in the queue is 10.81 seconds, the average time spent in the system is 111.05 seconds, and the server load is 10%.



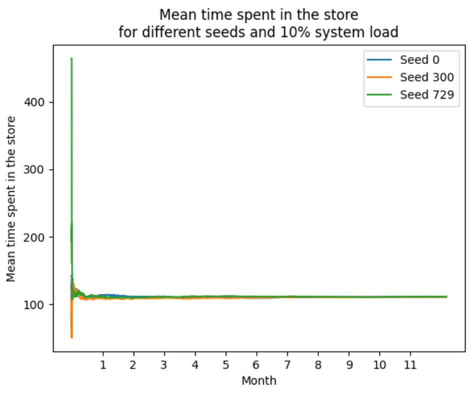
(a) Mean system length



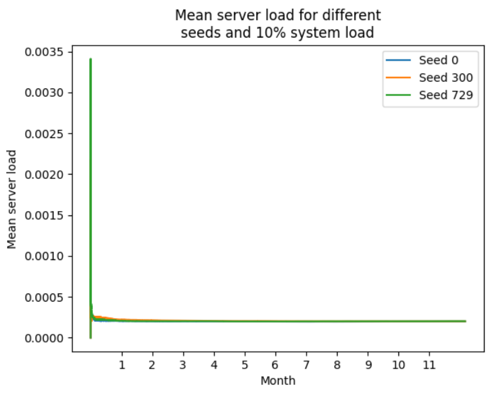
(b) Mean queue length



(c) Mean queue time



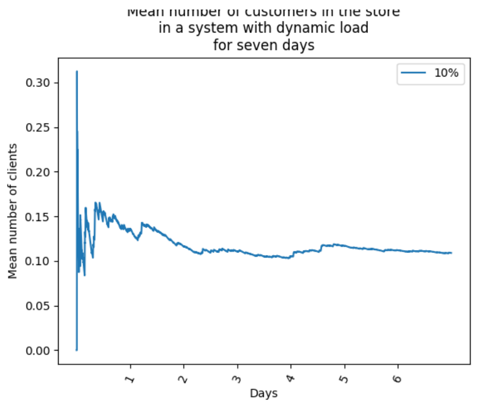
(d) Mean system time



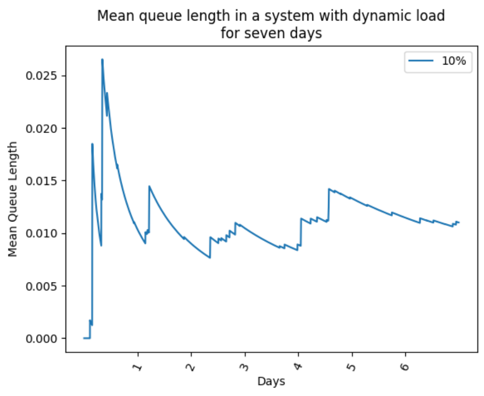
(e) Mean server load

Figure 3. Comparing key metrics at 10% server load across different simulation seeds.

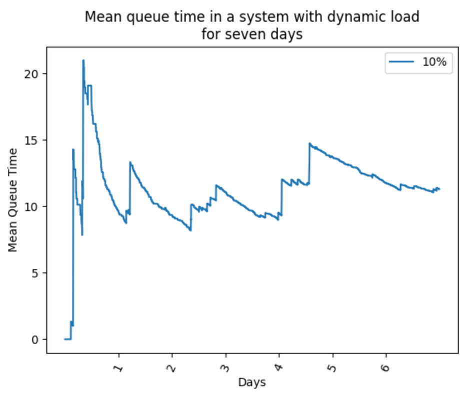
Figure 4 illustrates the same simulation over the first week. While a steady state isn't fully achieved, there's a noticeable trend approaching stabilization.



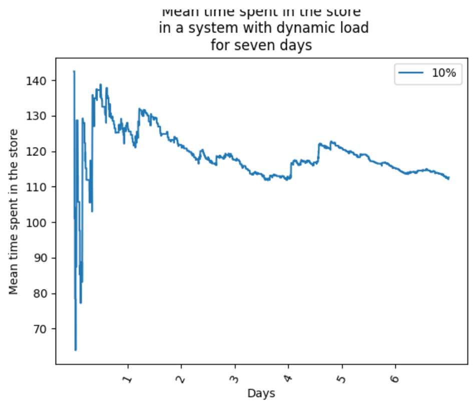
(a) Mean number of customers in the store



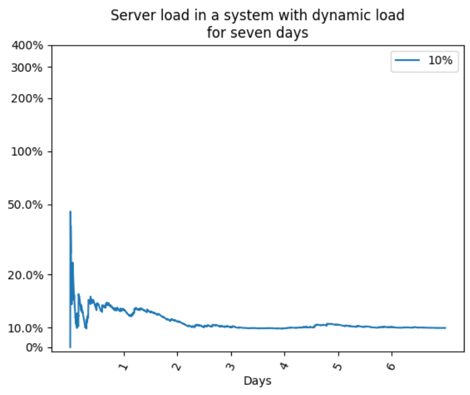
(b) Mean number of costumers in the queue



(c) Mean time spent in the queue



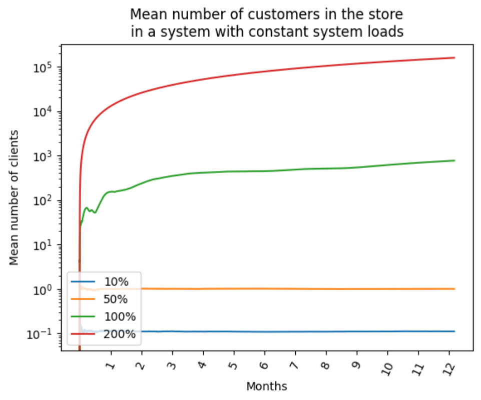
(d) Mean time spent in the store/system



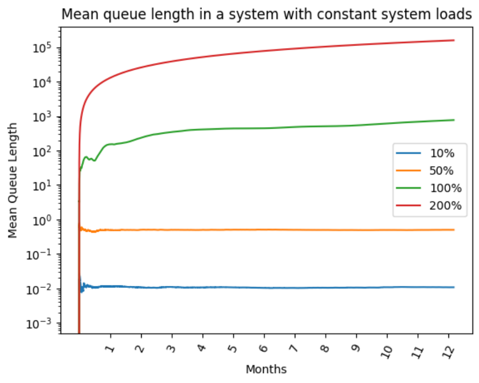
(e) Mean server load

Figure 4. 7-day short-term simulation to show the process of adjustmenting towards a stable state

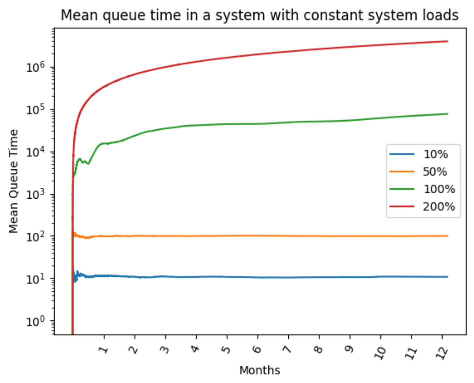
In figure 5 we can clearly see a stable state that is fully reached after a few months, both for the number of customers and for the time spent in the queue and in the system. This state is already reached after one month in the case of server load. A queueing system is considered stable when the average arrival rate exceeds the average service rate, i.e. fewer clients arrive than the server can handle. Once fully stabilised, each metric of the system converges to a given value. Looking at different constant system loads, we see that average arrival times that exceed the average service time tend to converge to certain values. Conversely, if the average service time equals or surpasses the average arrival time, we can observe indefinite growth in all metrics of the queuing system except the server load which converges at $serverload = \frac{servicerate}{arrivalrate}$. Obviously, even if the system load ist at 200% the server load can never be more than 100%. The prior meaning the system is stable, while the later leads to an unstable system as can be seen in Figure 5.



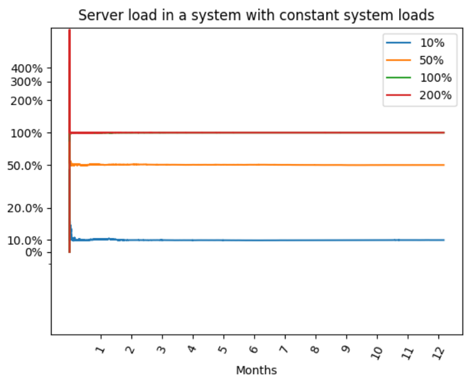
(a) Mean number of customers in the store



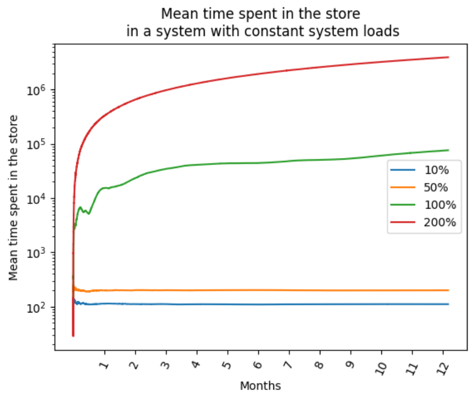
(b) Mean number of costumers in the queue



(c) Mean time spent in the queue



(d) Mean server load



(e) Mean time spent in the store/system

Figure 5. Comparison of main metrics at different call centre system loads.

3.3.2. Seasonality

In this section, we evaluate how the system responds to dynamically changing arrival times due to seasonality that occasionally exceed the average service time, making the system partly unstable.

Due to fluctuations in utilization across the year, it is necessary to dynamically adjust arrival times. The simulation assumes an equal number of days in each month for every year, disregarding leap years. Thus, specific timeframes where arrival rates change and their magnitude can be calculated selectively. Typically, a new client requires service approximately every 1000 seconds. However, during Christmas weeks, when the island's population surges twenty-fold, the arrival rate decreases by a factor of 20. Similarly, during summer months with a ten-fold population increase, the average arrival rate drops by a factor of 10. A drop in the average arrival rate by a specific factor indicates an increase in the number of clients compared to usual, therefore potentially making the system unstable, if the arrival rates are equal or smaller then the average service rate. The demand over the year is illustrated in Figure 6.

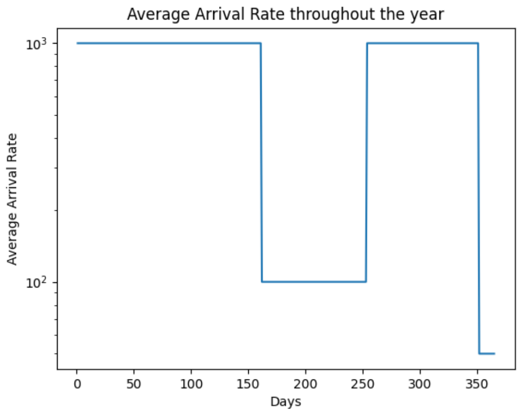
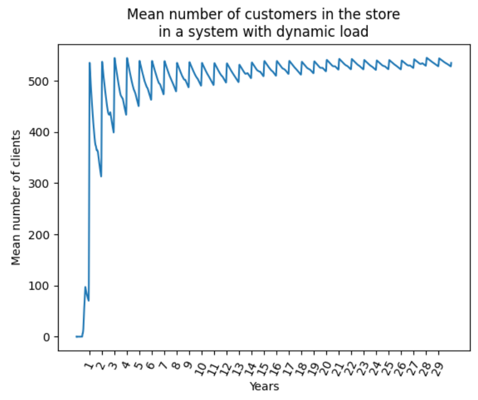


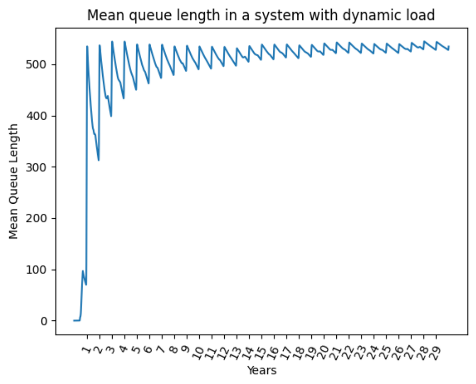
Figure 6. Simplified average arrival rate throughout the year, not accounting for leap years.

As previously discussed, when the service rate equals or surpasses the arrival rate of clients, the system becomes unstable. However, in this particular system, the average service time is 100 seconds. Considering the peak seasonal arrival times, we observe that during the summer, the arrival rate matches the service time. During Christmas, the service rate doubles the arrival rate, indicating that on average, two clients arrive for every service provided in that timeframe. Hence, the system exhibits partial instability. This phenomenon is depicted in Figure 7, where oscillations in values are evident. Values increase during unstable states or peak seasonal arrival rates in the simulation and decrease during typical arrival rates.

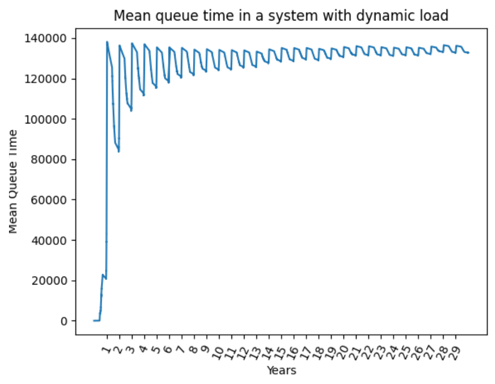
After simulating over a couple of years, we notice a trend where the system consistently approaches specific values across all metrics. The overall server load stabilizes at 40%, suggesting that a single satellite connection suffices to handle peak season demands and therefore no other satelite connection is needed. This stability implies that the system remains robust at its core, even amidst occasional partial instability. This assertion finds support in Figures 9e and 9f.



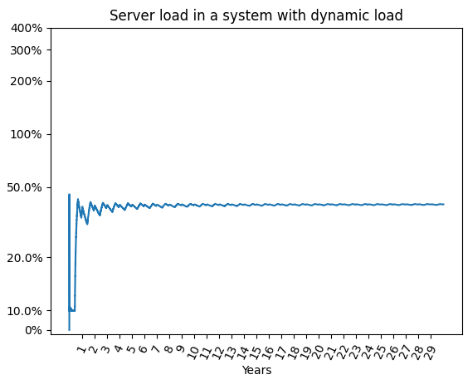
(a) Mean number of customers in the store



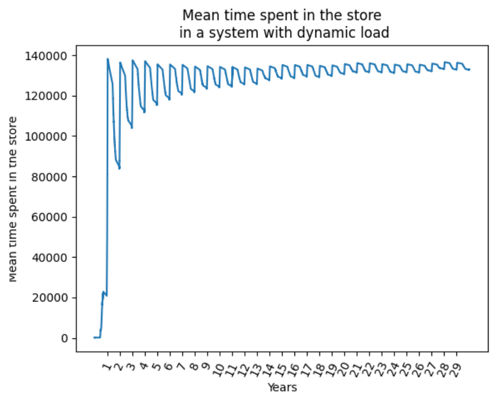
(b) Mean queue length



(c) Mean time a client spends in the queue



(d) Mean server load



(e) Mean time spent in the store/system

Figure 7. Simulating dynamic arrival rates.

3.4. Correctness of the Simulation

This section presents the tests we performed to prove the correctness of our simulation, specifically its input data with the Kolmogorov-Smirnov (section 3.4.1), its output data with the Little's Law Theorem (section 3.4.2), and its procedure with a custom made test (section 3.4.3).

3.4.1. Kolmogorov–Smirnov to test the Negative Exponential Number Generator

The Kolmogorov-Smirnov test (Listing 3) is a non-parametric test for equality of continuous one-dimensional probability distributions that can be used to determine whether a sample comes from a given reference probability distribution (one-sample K-S test) or whether two samples come from the same distribution (two-sample K-S test) [4]. In this case, the test is used to assess the fit of the generated store customer inter-arrival and service time data against a hypothesised exponential distribution. In fact, Java lacks native support for generating random numbers according to specific distributions beyond the more common ones such as constant or Gaussian. Therefore, we want to use the Kolmogorov-Smirnov (KS) test to test whether the *drawNegativeExponentialDistributedNumber* method implemented in our project actually generates random number data along a negative exponential distribution. As described in section 2.2, this method uses the java.util.Random library and the inverse transform sampling method.

```
from scipy import stats
import numpy as np

# Pseudo-randomly generated data
data = [1144.239911930156, 1698.5912040227304, 56.728264249446376, ...]

# Fit a negative exponential distribution and get parameters
params = stats.expon.fit(data)

# Generate expected values based on the fitted parameters
expected = stats.expon.cdf(np.sort(data), *params)

# Perform Kolmogorov-Smirnov test
statistic, pvalue = stats.kstest(np.sort(data), 'expon', params)
print("Kolmogorov-Smirnov␣Test:")
print("Statistic:", statistic)
print("P-value:", pvalue)
# Interpret the result fot Significance level = 0.5
sig_level = 0.05
if pvalue > sig_level:
    print("The␣data␣follows␣an␣exponential␣distribution.")
else:
    print("The␣data␣does␣not␣follow␣an␣exponential␣distribution.")
```

Listing 3: Performing the Kolmogorov-Smirnov Test using python and scipy

This test can be found in the file *KolmogorovSmirnovTest.py*. It uses as input a sample of the negative exponential inter-arrival and service times generated in our simulation and as parameter a significance level α of 5%. This corresponds to the risk of Type I error - the probability of rejecting the null hypothesis when it is actually true. As shown in theListing4, the output of the KS test gives a p-value of approximately 0.935. As this p-value is much greater than the significance level, we cannot reject the null hypothesis. This result indicates that there is no significant statistical evidence to conclude that the data does not follow the hypothesised negative exponential distribution, suggesting that the sample data is consistent with the specified distribution.

```
Kolmogorov-Smirnov Test:
Statistic: 0.016827614936299307
P-value: 0.9350078966105504
The data follows an exponential distribution.
```

Listing 4: Results of the Kolmogorov-Smirnov Test

In addition, the inter-arrival and service data are plotted in *plots.py* (Fig. 8), confirming adherence to the negative exponential distribution pattern, and their average is calculated for a 10% call shop load in *Statistics.py* (Listing 5), confirming consistency with the expected averages of 1000s and 100s set for the simulation.

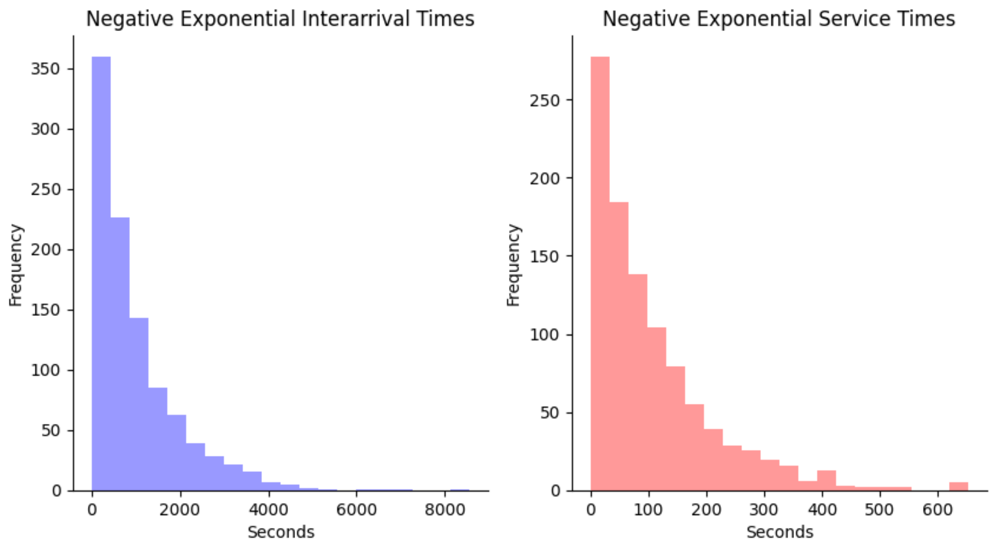


Figure 8. The Probability Density Function (PDF) of Negative Exponential Distributions: A Comparison between an average of 1000s (in Blue) and 100s (in Red).

```
After a year the mean time spent at the server: 100.24236124140205
After a year the mean arrival time: 1002.7206349348213
```

Listing 5: Computed average inter-arrival and service time for a shop simulation with 10% load

3.4.2. Little's Law to test the Simulation Procedure

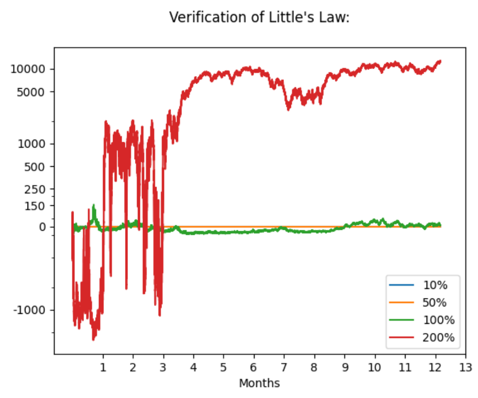
Little's law is a central law of traffic theory. "[It] states that the long-term average number L of customers in a stationary system is equal to the long-term average effective arrival rate λ multiplied by the average time W that a customer spends in the system" [5]. This is equivalent to [3, p. 216]

$$L = \lambda w \tag{1}$$

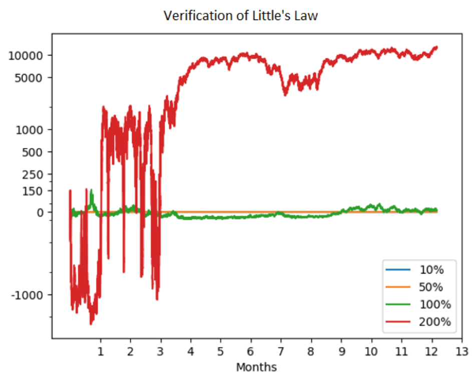
, or

$$Q = \lambda d \tag{2}$$

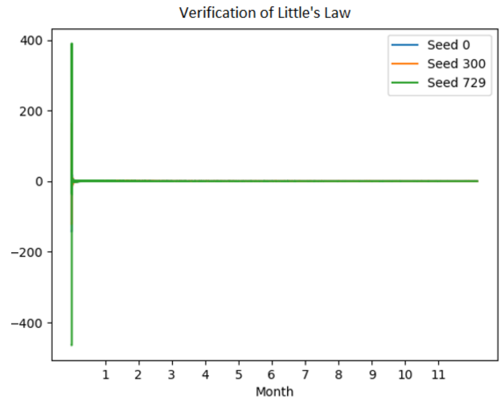
, whereby Q is the mean number of clients in the queue, d the mean time spent the queueu, L the mean number of clients in the system, w the mean time spent in the system, and λ the arrival rate in the system, each in steady state. The metrics mentioned are computed during simulation and outputed in the file *Statistcs.py*, allowing us to verify simulation correctness by checking convergence against 0 in a stable system. In a stable system, both sides of the equation should be equal, so we can calculate the difference between the product of λW and L, as shown in Figure 9



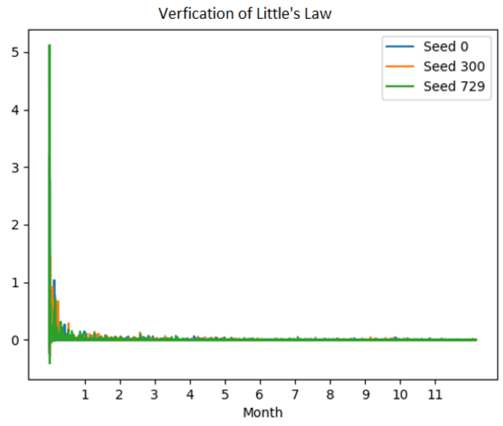
(a) Little's Law (based on queue data) converging to 0 for different server loads



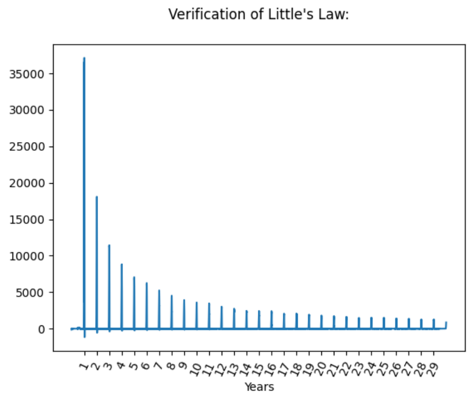
(b) Little's Law (based on system data) converging to 0 for different server loads



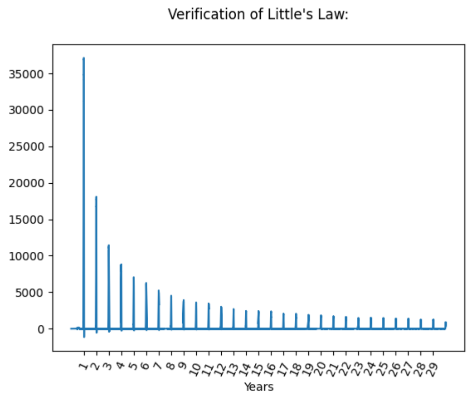
(c) Little's Law (based on queue data) converging to 0 for different seeds of a 10% load



(d) Little's Law (based on system data) converging to 0 for different seeds of a 10% load



(e) Little's Law (based on queue data) converging to 0 with season-based inter-arrival times



(f) Little's Law (based on system data) converging to 0 with season-based inter-arrival times

Figure 9. Verification of the simulation using Little's Law

3.4.3. Manual Simulation Test

The small custom test we implemented aims to verify that our simulation is handling events correctly. This can be found in the file *SimulationTest.java*.

It follows the same model and rules described in this chapter. The only difference to the main script is that it uses predefined values for customer arrival and service times, for which we have calculated the expected output by hand. This allows us to compare the results and check that each event in the simulation has been handled correctly by the code.

The arrival and service times have been chosen to cover all three possible scenarios that can arise in our model and are respectively 2.0, 4.0, 1.0, 2.0, 1.0, 1.0, 1.0, 4.0, 10.0 and 3.0, 1.0, 6.0, 0.1, 0.1, 7.0, 0.1, 2.0, 10.0. The output shows the interval j, queue length qL, server length sL, system length $sysL$ for each event. After the symbol ||, also the arrival times of the clients waiting in the queue (represented by the first []) and the calculated end of service for the client standing at the server (represented by the second []) are visualised. These results are correct and show that the simulation is able to handle all scenarios correctly.

```
j: 0.0-0.0; qL: 0.0; sL: 0.0; sysL: 0.0
j: 0.0-2.0; qL: 0.0; sL: 0.0; sysL: 0.0 || [] [5.0]
j: 2.0-5.0; qL: 0.0; sL: 1.0; sysL: 1.0
j: 5.0-6.0; qL: 0.0; sL: 0.0; sysL: 0.0 || [] [7.0]
j: 6.0-7.0; qL: 0.0; sL: 1.0; sysL: 1.0 || [] [13.0]
j: 7.0-9.0; qL: 0.0; sL: 1.0; sysL: 1.0 || [9.0, ] [13.0]
j: 9.0-10.0; qL: 1.0; sL: 1.0; sysL: 2.0 || [10.0,9.0, ] [13.0]
j: 10.0-11.0; qL: 2.0; sL: 1.0; sysL: 3.0 || [11.0,10.0,9.0, ] [13.0]
j: 11.0-12.0; qL: 3.0; sL: 1.0; sysL: 4.0 || [12.0,11.0,10.0,9.0, ] [13.0]
j: 12.0-13.0; qL: 4.0; sL: 1.0; sysL: 5.0 || [12.0,11.0,10.0, ] [13.1]
j: 13.0-13.1; qL: 3.0; sL: 1.0; sysL: 4.0 || [12.0,11.0, ] [13.2]
j: 13.1-13.2; qL: 2.0; sL: 1.0; sysL: 3.0 || [12.0, ] [20.2]
j: 13.2-16.0; qL: 1.0; sL: 1.0; sysL: 2.0 || [16.0,12.0, ] [20.2]
```

Listing 6: Testing with a costum made test

4. Discussion and Conclusion

In conclusion, this study has successfully achieved its objectives of simulating constant and dynamic arrival rates within an M/M/1 system and an event-based approach. Through the application of Little's Law, positive results were obtained, indicating the successful implementation of the simulation, which converges to zero as expected. Furthermore, the analysis of seasonal arrival times showed that the server load stabilises at around 40%, suggesting that the call shop is under-utilised, negating the need for a secondary satellite link. Even with partially unstable conditions, Little's Law still yields favourable results, underlining the overall stability of the system despite dynamically changing inter-arrival times that fall below the average service rate. Verification of the negative exponential distribution using the Smirnov-Kolmogorov test showed successful validation of the random numbers generated. Taken together, these results confirm the reliability of the simulated M/M/1 system.

Future research might explore when the simulation begins to lose stability. This could include gradually increasing the number of days with seasonal peaks to see at which point the simulation becomes unbalanced. From a coding perspective, updates could help make the scripts easier to use and more broadly applicable. For other users, grasping how the Java and Python components of your projects interlink might be challenging. Furthermore, the test implemented in *SimulationTest.java* is currently tailored to certain data inputs.

References

1. Class Random, 2024.
2. Inverse transform sampling, 2023.
3. Bungartz, H.J.; Zimmer, S.; Buchholz, M.; Pflueger, D.; Borne, L.S.; Borne, L.R. *Modeling and simulation an application-oriented introduction*; Springer Berlin, 2016.
4. Kolmogorov–Smirnov Test. In *The Concise Encyclopedia of Statistics*; Springer New York: New York, NY, 2008; pp. 283–287. https://doi.org/10.1007/978-0-387-32833-1_214.
5. Little's law – Wikipedia, 2024.