Monte Carlo Simulations and Saving Murphy

Lucas Cho, Jonathan Della Santina, Javier Garcia,
Theo Holmes, Bernie Liu, Vladimir Radostev
University of Washington, Department of Mathematics, Seattle, Wa, 98195

June 5th 2024

Abstract

Murphy's Pub is an Irish restaurant bar looking to increase profits to avoid their building being sold. Management has noticed that a central limiting cost of running the business is non-optimized labor scheduling. We partnered with the owners of Murphy's Pub to use their amassed sales data from their Point-of-Service system to predict average hourly customer flow throughout the days of the week. We then implement a Monte Carlo simulation to model how service speed responds to server quantity. Using the results generated from the simulation, we create a weekly schedule that schedules the fewest servers necessary to fulfill management's constraints of customer satisfaction rates. The end result is an optimized scheduling look-up chart for an average week that holds the minimum number of servers Murphy's Pub needs to satisfy customers.

1 Introduction

1.1 Murphy's Pub

Murphy's Pub is Seattle's first Irish pub, first opened on May 18, 1981, a year after the Mt. St. Helen's explosion. On February 12th, 2015, Murphy's reopened under the ownership of husband and wife Phil and Chelley in Wallingford, Washington. Their current business model has reportedly kept the establishment sustainable but due to the COVID-19 pandemic, recovering past profits has proved difficult. Additionally, the prospect of competitors looking to convert the land into profitable housing threatens Murphy's Pub with eviction. The combination of these factors has motivated the owners of Murphy's to seek alternative methods to increase profit without raising food prices. Specifically, adopting a scheduling model that minimizes wasted labor costs while maintaining quality of service.

1.2 Problem Objectives:

Murphy's must create a schedule that reflects demand and minimizes average labor costs while maintaining quality of service. Currently, Murphy's management uses personal experience to schedule the servers needed for each day of the week. The current intuition-based scheduling has been shown on average to secure additional revenue during high-demand days such as on weekends or during sporting events. However, this is not always the case during "slow" days. Generally, the high quality of service standard tends to over-prepare in case of a rush, while consistently paying for the presence of servers even during slow hours. To begin addressing this issue we begin by trying to answer the following question. What is the minimum number of servers we need to be scheduled during each day of the week to satisfy a percentage of customer requests within a certain number of minutes? After multiple meetings with the owners, and interviewing patrons of the pub, figuring out the minimum number of servers required to attend 90% of customer requests within 5 minutes became the goal.

2 Scheduling

2.1 Serving staff and Customers

The serving staff at Murphy's Pub consists of bartenders, barbacks, and servers. Barbacks generally focus on their duties and typically do not take customer orders, make drinks, or conduct payments. Bartenders work the bar during the majority of their shift, however, if work demand is low or when the pub is understaffed, they also assume the aforementioned responsibilities central to serving customers. Consequently, we lump servers and bartenders together under the single title of "server" for the remainder of the paper.

2.2 Data-Based Scheduling

Like the majority of businesses today, Murphy's has continuously collected and stored relevant sales data for potential future analysis using a Point-of-Service system (P.O.S.). Using the data collected by the P.O.S., we sought to predict the flow of customer demand during an average day of the week to also predict the total service staff required to meet the satisfaction goal. We expect that the combination of human and data-based prediction will create a more systematic way of scheduling service staff and thus minimize labor costs.

3 Mathematical Modeling

3.1 Literature Overview

Simulation models are widely used in operational research to optimize resource allocation and improve service efficiency, particularly in contexts involving queuing theory. Monte Carlo simulations are a prominent tool within this domain, effectively modeling random processes to predict outcomes. The paper "Optimization of Food Industry Production Using the Monte Carlo Simulation Method: A Case Study of a Meat Processing Plant" explores how Monte Carlo simulations can be used to optimize production processes and served as partial motivation for applying Monte Carlo simulations in the context of this problem [2]. The study presents a model to automate budgeting and management decisions in a meat processing plant, demonstrating a significant increase in profits by more than 30 percent through the application of Monte Carlo methods. Another study titled "Restaurant closures during the COVID-19 pandemic: A descriptive analysis" also described the utilization of Monte Carlo simulations to forecast daily meal consumption and optimize inventory management in restaurants [3]. The study highlights how this approach helps in reducing waste and improving service efficiency by accurately predicting customer demand and aligning resource allocation accordingly. By integrating Monte Carlo simulations into their operational strategy, Murphy's Pub can enhance its scheduling strategies, improving service quality and profitability through more efficient resource allocation. Before discussing Monte Carlo simulations in greater detail, some background on probability theory and queuing theory is warranted.

3.2 Probability Theory and Queuing Theory

Probability theory is a branch of mathematics that studies uncertainty to model and describe random events. To do so, core concepts such as random variables and probability distributions are defined. A random variable is a numerical representation of the outcomes of a random event. For example, imagine rolling a six-sided die. The result of the die roll is a random variable because it can vary every time you roll, but it's still one of those six specific numbers. A probability distribution describes how probabilities are assigned to these possible outcomes. For instance, the die from the previous example obtains what is called a uniform probability distribution assigning each outcome (1 through 6) a probability of 1/6. Continuous random variables, on the other hand, might describe measurements like temperature or time, and their distributions are often represented by probability density functions (PDFs) that show the likelihood of different outcomes within a continuous range.

Queuing theory, as the name suggests, is the study of waiting lines or queues. By defining and analyzing various aspects of queues, such as arrival rates and service rates, queuing theory helps design and manage systems efficiently to minimize wait times and improve service. In our model, we use two different queues. One of the queues is designated towards customers arriving at the pub and wanting tables when they are none available. The other queue is dedicated to requests made by customers when there are no servers available.

3.3 Monte Carlo Simulations

A Monte Carlo simulation is a computational technique used to understand the behavior of complex systems and processes by using random sampling. Named after the famous casino in Monaco, this method relies on repeated random sampling from probability distributions to obtain numerical results [4]. We chose to use Monte Carlo simulations to model service at Murphy's Pub due to the random nature of customer arrival rates and the potential to predict customer arrival rates from the data collected by Murphy's P.O.S. system. The following aspects were modeled through random distributions:

- Time between customer arrivals
- Time for a server to complete a request
- Duration of each customer's stay

3.4 Data Collection and Analysis

Like many restaurants, Murphy's Pub utilizes a P.O.S. (Point of Service) system to be able to efficiently track and digitize customer tabs, deliver order requests to the kitchen, and streamline payments. Many of these digital services already store customer receipts within a database out of legal necessity in case of future financial disputes, with the bonus of providing sales insights for management. Murphy's Pub uses a P.O.S. called SpotOn, which records the digital receipts of each past closed ticket. These records contain lots of useful information in addition to payments.

In addition to adapting sales data, we collected data at the Pub for information not captured by the P.O.S history. This includes:

- Quantity of customer requests
- Time to satisfy those requests

The most important insights we wanted from the data for our model were the following questions:

- For each day of the week, at what rate do customers arrive?
- How long do customers stay?

Analysis was done on exported .csv files containing nearly all tickets opened between 03/01/24 - 5/17/24. Statistical outliers such as St. Patrick's Day were excluded because of their significant skew on demand predictions, and these days already consist of a full staff roster so no prediction is necessary. This range was chosen in case seasonality was a factor that influenced customer flow. No data before 1/13/2024 was collected on the SpotOn system and converting data stored on the previous system promptly proved impractical. Each .csv file contains entries for all tickets opened that day, the following relevant attributes were used:

- 'Open' - representing the time of day each ticket was opened
- 'Close' - representing the time of day each ticket was closed/paid for
- 'Items' - the amount of food or drink items on the tab before the tab was closed
- 'Time Open' - representing the total time the ticket was open
- 'Table Number' - representing the table number of the ticket

We began by pooling together each day of the week. This allowed us to get a general sense of what each day of the week looked like. We then measured the number of tickets opened during each hour of each day of the week. We then split this data into tickets with a table number, and tickets without a table number since tickets without a table number correlated to items ordered at the bar. We divided each of these quantities by the number of datasets combined for that day of the week. This gave us accurate average approximations for customer arrival rate, and proportion of customer types. The customer types that were found, and proved significant enough to distinguish, were "table-goers" and "bar-goers". As the names suggest, table-goers are customers who enter the pub seeking to sit at a table or a booth, while bar-goers are customers who tend to sit or stand near the bar and order from there during the majority of their stay. Due to the tendency to close out a ticket every time a drink is ordered, bar-goer data had to be separated so that a reasonable average length of stay could be calculated for customers who order from tables.

To calculate the general arrival rate of customers we consider the following histograms of data:

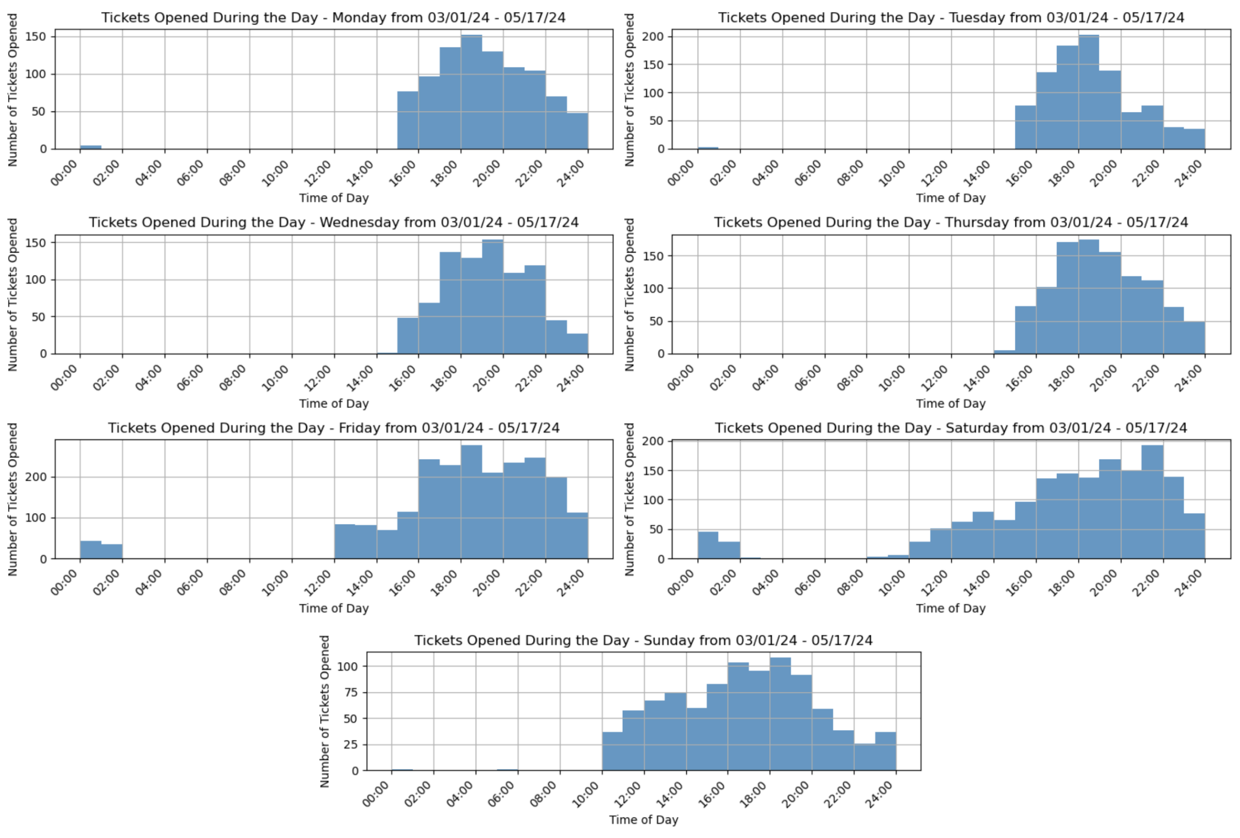
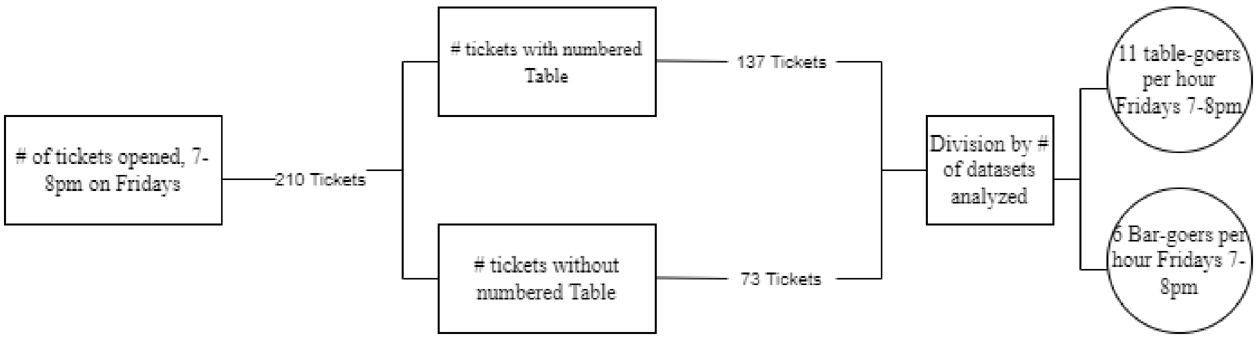


Figure 1: Number of tickets opened on average per hour per day

(Note: Histograms contain the combined number of tickets opened over multiple Fridays, Mondays, etc. Some histograms measure tickets opened over more days than others)

Then for each hour of each day, we applied the calculation described in Figure 2.



# of tickets opened, 7-8pm on Fridays
210 Tickets
# tickets with numbered Table
137 Tickets
# tickets without numbered Table
73 Tickets
Division by # of datasets analyzed
11 table-goers per hour Fridays 7-8pm
6 Bar-goers per hour Fridays 7-8pm

Figure 2: Example calculation for 7-8pm arrival rate on Friday using 12 datasets

Referring to the mathematical definition of inter-arrival time proposed in [4], we can use approximations for average arrival rate to create the probability mass function for the time until a new customer arrives.

$$f(t) = \lambda e^{-\lambda t}$$

where λ is the mean arrival rate at the current hour of the current day, and t corresponds to minutes. Here, random samples below the curve will accurately sample realistic times between customer arrivals. We can better visualize and understand this function with it's Cumulative Density Function to approximate the probability a customer arrives within x minutes after a previous party

$$F(x) = \int_0^x \lambda e^{-\lambda t} dt$$

$$F(x) = 1 - \ e^{-\lambda x}$$

Here the function models the probability that a customer arrives within x minutes or less following the previous customer.

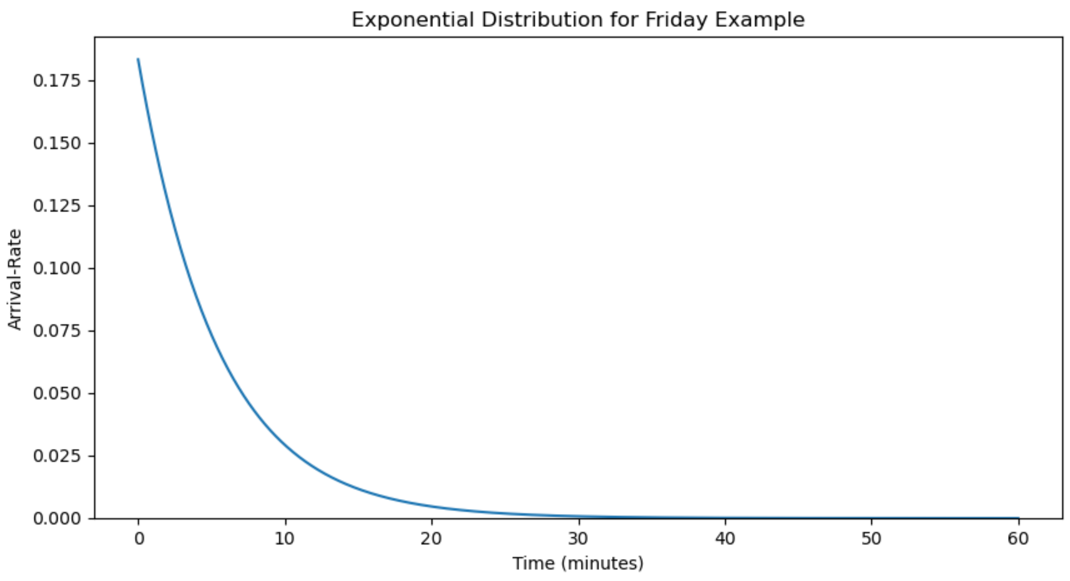


Figure 3: Exponential Distribution for Table-goer arrival rate 7-8pm Fridays.

To predict how long each customer that arrives stays, we created a discrete probability distribution using data on how long each ticket is open

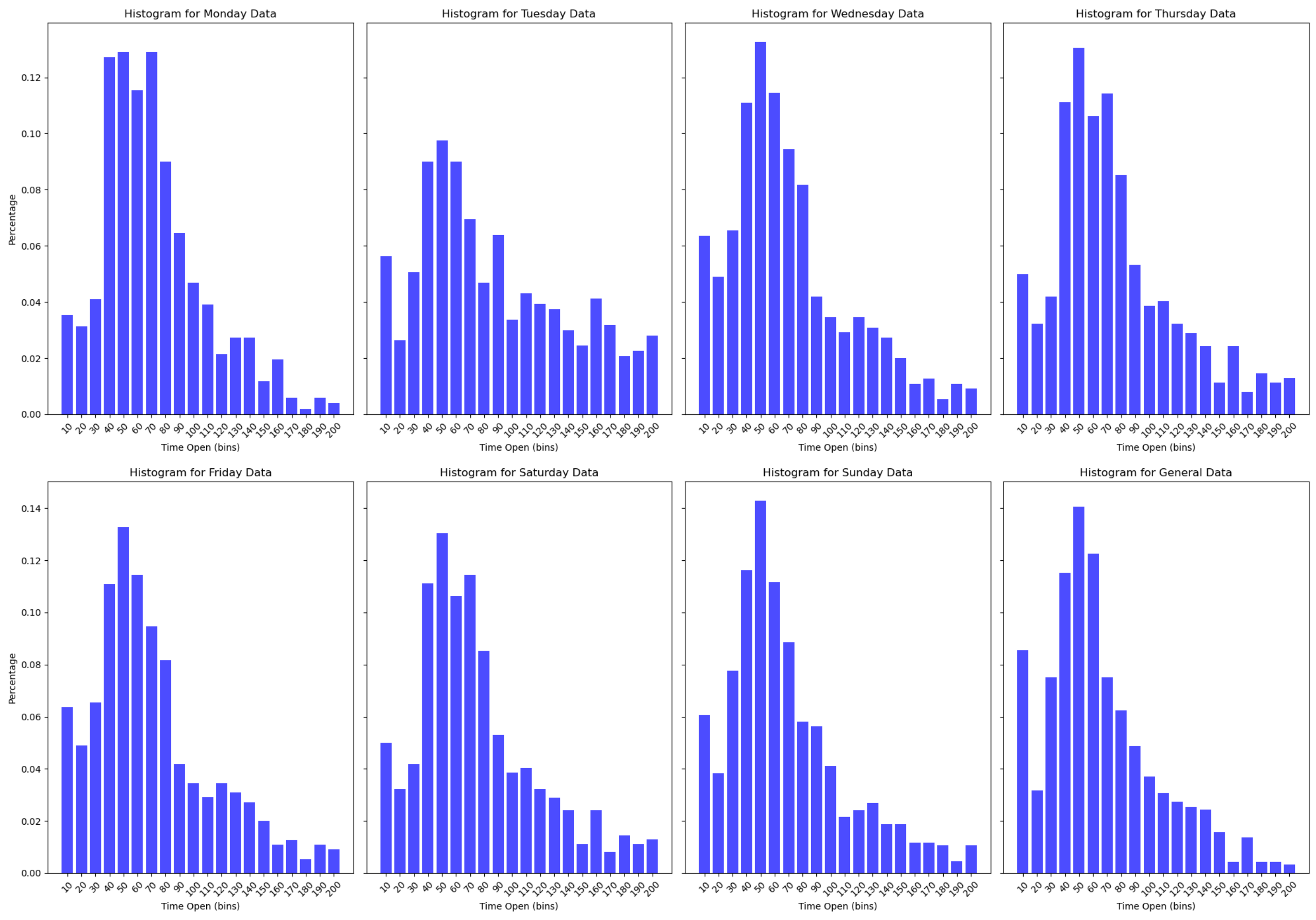


Figure 4: Ticket life spans for each day of the week

4 Mathematical Assumptions and Simplifications

Due to the large variety of real-world variables present, simplifications and assumptions were necessary to feasibly model our simulation and obtain practical results. Below is a list of the simplifications and assumptions made:

- Since a single ticket is opened for groups of customers that sit together, groups are treated as a single customer in our model. Essentially this means that we are modeling tickets, and not people, which is still a somewhat accurate representation of people in the pub.
- Balking was not considered in the model. This means that customers who arrive are going to stay regardless of the wait for tables. Since all of our customer arrivals are based on tickets, this assumption makes the simulation account for the correct number of customers per day regardless of the wait.
- Every customer that enters has a predetermined time until departure. They will then make requests until their departure time. We assume this because from our observations, this is essentially what customers are doing in the pub.
- Customers occupy at most one table. While a customer could occupy more than one, it is rarely prevalent in the pub, and there is not enough information to model this. This makes our model slightly inaccurate.
- Although bartenders and servers generally have different responsibilities, their ability to assume each other's responsibilities during rushes imply that their combined labor output is not significantly bottle-necked by their specific roles.
- Table-goers are constrained by table quantity (16) while we assume unlimited bar capacity for bar-goers. This is because customers at the bar tend to stand around if the seats are full.
- Idle servers immediately serve the next waiting customer. Since we are minimizing labor costs, especially during peak hours, we should model servers that are constantly working.

5 Simulations

To formulate our model we drew inspiration from the general construction of a simulation model for a single-server queuing system from the [4]. However, our model requires us to have multiple servers and other extra considerations that the model did not consider.

The model can be described in four parts, a customer arrival event, a request event, a request finish event, and a departure event. Upon running the simulation, a customer instance is created. We immediately generate a random predetermined length of stay according to ticket duration data, and a random time until the next customer arrives according to the exponential distribution modeled using ticket opening rate data. If the customer is a bar-goer, they are ”seated” immediately since we’ve assumed unlimited seating for the bar. If the customer is a table-goer a query is made on whether there is a table available. If there is a table available, the customer is seated. If there isn’t, they enter the table queue.

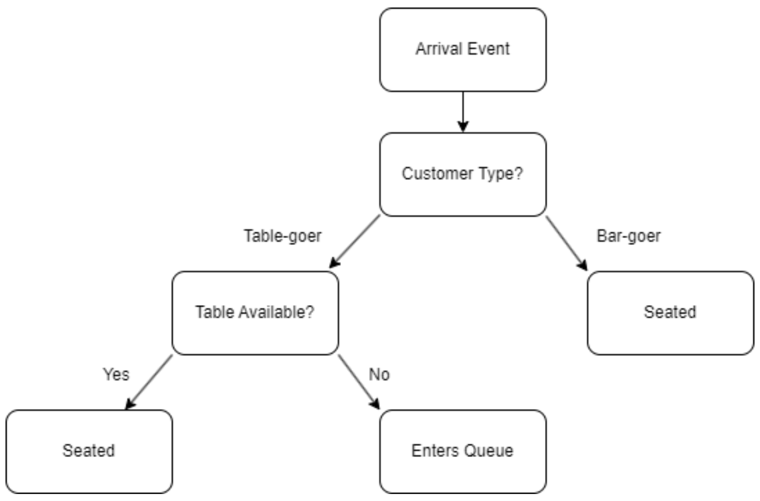


Figure 5: Customer Arrives at Pub

Once a customer is seated they make their first request. A query is made on whether a server is available. If so, the customer’s request is attended to and the number of servers decrements by one. If not, they enter a request queue, waiting for a server to become available to process their request. If the customer waits longer than five minutes in the queue, they are marked unsatisfied. This process repeats for every subsequent request that the customer makes.

When a server begins fulfilling a request, they become occupied with the request for a duration sampled from related recorded data.

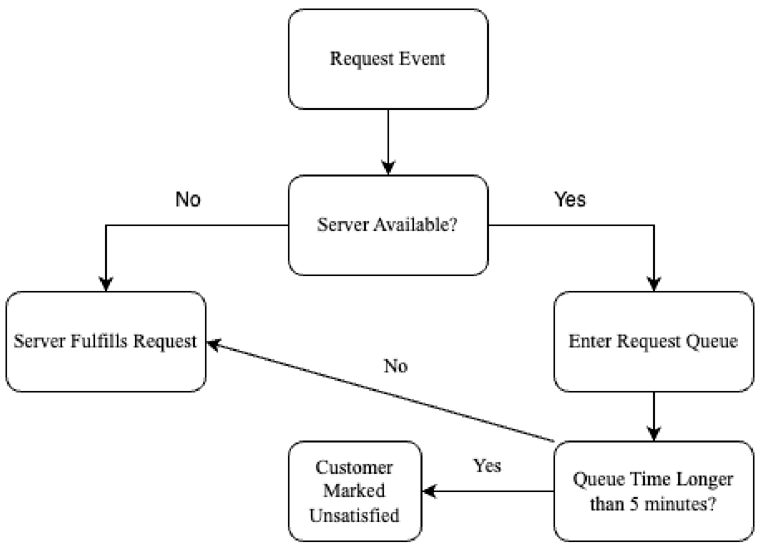


Figure 6: Customer makes a request

When a customer’s request is fulfilled by a server, they choose to either stay and make another request or leave at their pre-determined departure time. In the model, if a customer determines they can make a request before their departure time, they will make another request, otherwise they will depart at their departure time. For servers, once a request is completed, they check to see if there is another request in the waitlist. If there isn’t, the number of available servers increment by one, and if there is, they attend to that request.

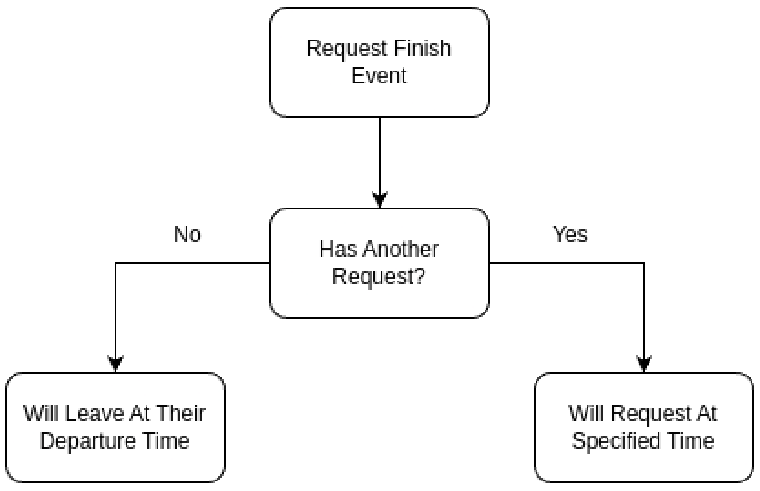

Request Finish Event
Has Another Request?
No
Yes
Will Leave At Their Departure Time
Will Request At Specified Time

Figure 7: Customer Request is Finished

When a customer at the bar departs, nothing changes. However, when a customer sitting at a table departs, if there was another customer in the table waitlist, then they are seated to that table and make a request. If no one was waiting for a table, then the number of available tables increments by one.

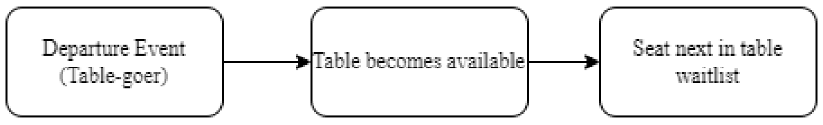


Figure 8: Table-goer departure event

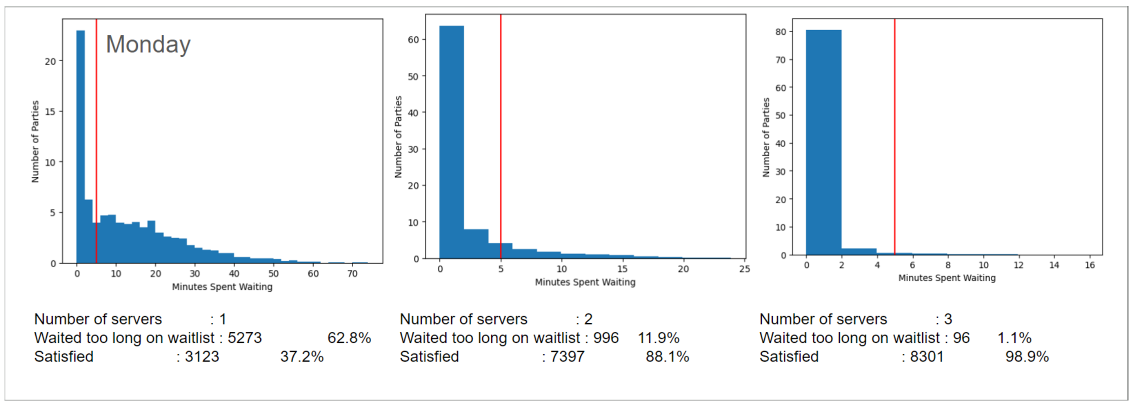
6 Results and Analysis

We ran simulations for each day of the week 1000 times for the duration that Pub was open in minutes on the simulated day with the following inputs:

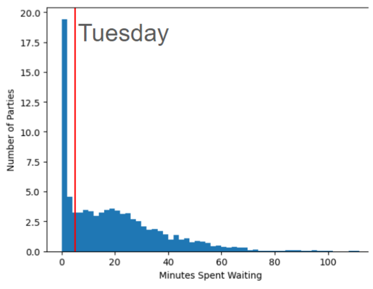
- Number of servers
- Number of tables (16 tables)
- Figure 1 Histogram data for the corresponding day (e.g. if Monday was simulated then only Monday data was inputted)
- Figure 4 Histogram data
- Service time data collected on site
- Time pub is open in minutes

We ran our simulation to find average satisfied vs. non-satisfied customers on a given day and computed the percentages accordingly. Three different server numbers were tested, 1 server, 2 servers, and 3 servers.

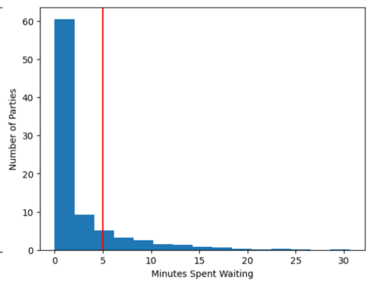
These were the final results:



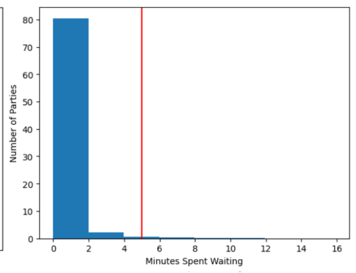
```
Number of servers          : 1
Waited too long on waitlist : 5273          62.8%
Satisfied                  : 3123         37.2%

Number of servers          : 2
Waited too long on waitlist : 996    11.9%
Satisfied                  : 7397         88.1%

Number of servers          : 3
Waited too long on waitlist : 96     1.1%
Satisfied                  : 8301         98.9%
```



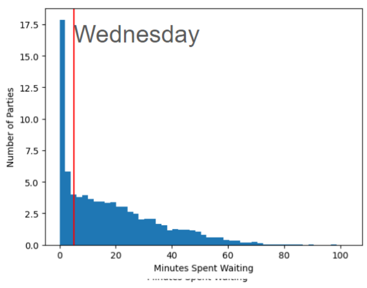
Number of servers : 1
Waited too long on waitlist : 5972 70.0%
Satisfied : 2564 30.0%



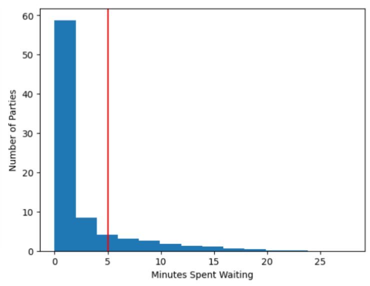
Number of servers : 2
Waited too long on waitlist : 1326 15.5%
Satisfied : 7223 84.5%



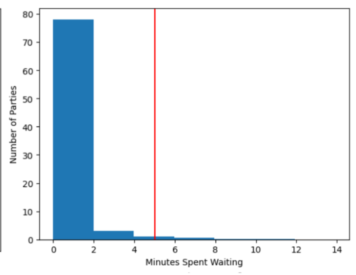
Number of servers : 3
Waited too long on waitlist : 271 3.2%
Satisfied : 8227 96.8%



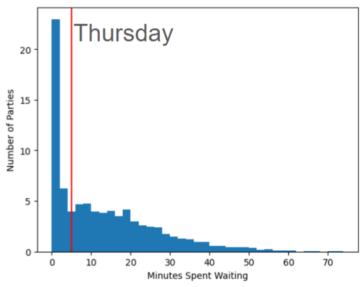
Number of servers : 1
Waited too long on waitlist : 5948 69.9%
Satisfied : 2561 30.1%



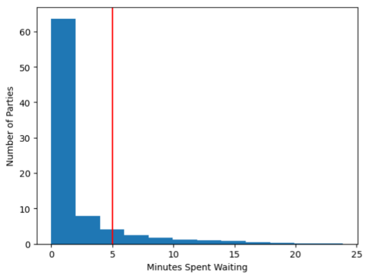
Number of servers : 2
Waited too long on waitlist : 1324 16.0%
Satisfied : 6951 84.0%



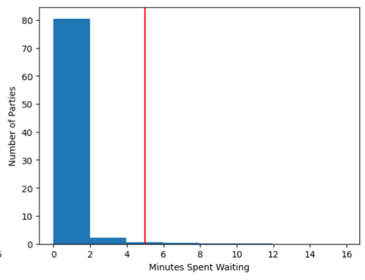
Number of servers : 3
Waited too long on waitlist : 96 1.1%
Satisfied : 8301 98.9%



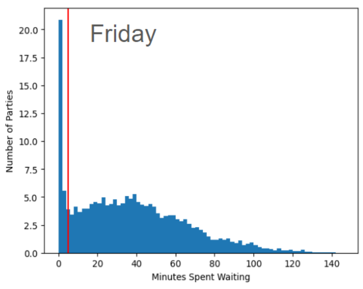
Number of servers : 1
Waited too long on waitlist : 5408 63.8%
Satisfied : 3069 36.2%



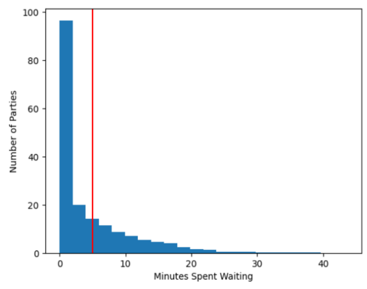
Number of servers : 2
Waited too long on waitlist : 1085 13.0%
Satisfied : 7281 87.0%



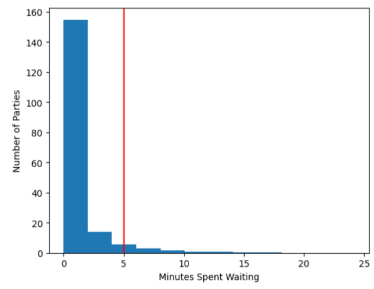
Number of servers : 3
Waited too long on waitlist : 128 1.5%
Satisfied : 8406 98.5%



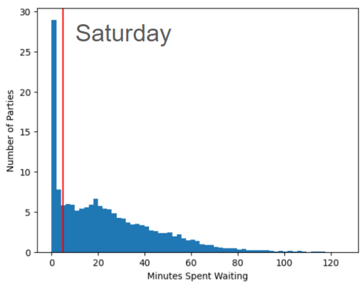
Number of servers : 1
Waited too long on waitlist : 15221 84.3%
Satisfied : 2841 15.7%



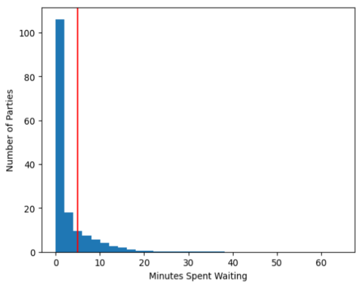
Number of servers : 2
Waited too long on waitlist : 5513 30.7%
Satisfied : 12416 69.3%



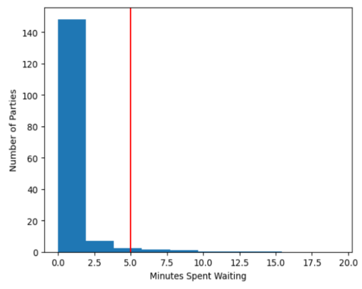
Number of servers : 3
Waited too long on waitlist : 911 5.0%
Satisfied : 17178 95.0%



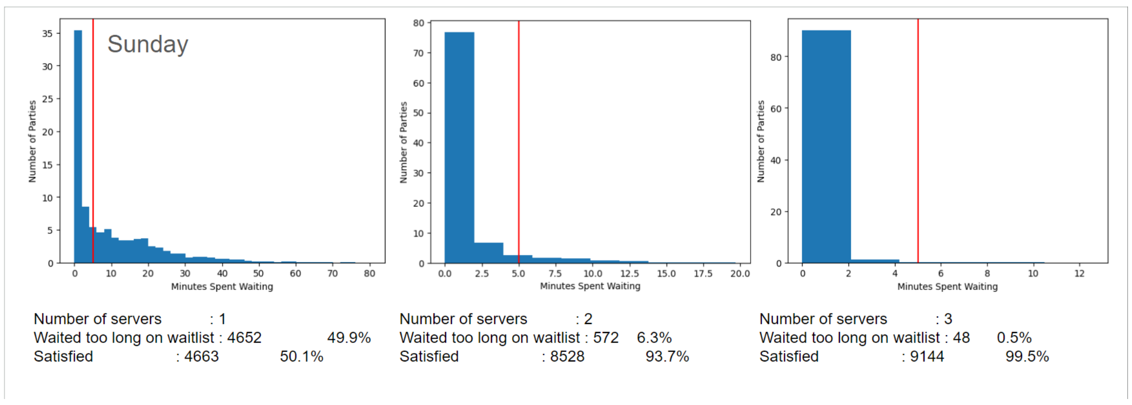
Number of servers : 1
Waited too long on waitlist : 12116 75.3%
Satisfied : 3966 24.7%



Number of servers : 2
Waited too long on waitlist : 3058 19.2%
Satisfied : 12893 80.8%



Number of servers : 3
Waited too long on waitlist : 410 2.5%
Satisfied : 15721 97.5%



```
Number of servers          : 1
Waited too long on waitlist : 4652          49.9%
Satisfied                  : 4663          50.1%

Number of servers          : 2
Waited too long on waitlist : 572    6.3%
Satisfied                  : 8528         93.7%

Number of servers          : 3
Waited too long on waitlist : 48     0.5%
Satisfied                  : 9144         99.5%
```

It is readily apparent that three servers comfortably satisfy over 90% of customers within 5 minutes, and this reflects our onsite observations when monitoring the pub with three servers. When it came to 2 servers some discrepancies needed to be investigated. Consider Friday and Saturday, the two busiest days at the pub by far, which also explains why having two servers on those days led to the lowest satisfaction percentages. However, despite being similarly busy, there is an over 11% difference in satisfaction percentage with two servers. This turned out to be a result of the sharp spike in arrival rate on Fridays compared to Saturdays. Referring to Figure 1, it can be seen that after 5 pm the total number of tickets opened more than doubles from just over 100 tickets for the hour to over 250 tickets vs. Saturday which does not see a change in tickets opened greater than 50. This sharp increase in tickets opened, which to us reflected customer arrival rate, meant more customers were arriving more frequently, thus filling up the request queue rapidly. This in turn meant more and more customers were waiting over 5 minutes to get their requests fulfilled which led to lower satisfaction percentages. Saturday on the other hand has a histogram of tickets opened per hour that almost resembles a continuous function. This ”smoother” or less sharp increase in tickets opened per hour meant the request queue was filling up in a more manageable manner as opposed to filling up all at once. This same explanation can be used to explain the difference in satisfaction percentage for Monday and Wednesday for example. Furthermore, this also gives some insight into when scheduling a third server may be most beneficial such as on Fridays after 5 p.m..

Formal variables are defined in the appendix.

7 Future Directions

Since there are an immense number of factors that need to be considered when modeling a restaurant, some key features were left out of the final model:

- Allow customers to take up multiple tables.
- Differentiate between types of requests (drink refills, more food, etc).
- Create a departure queue to simulate waiting to pay a tab
- Create a table cleaning queue (tables should be cleaned before they are marked available)
- Allow shifts for servers. Currently our model doesn't allow a server to clock in or clock out because we are only trying to measure the number of servers needed on a particular day of the week, not the quantity needed at each hour of every day.
- Simulate server multi-tasking
- Incorporate outlier days such as St. Patrick's Day or game days.
- Implement the kitchen processes. This would make the model encapsulate nearly the whole restaurant to help describe any sort of event occurring at the pub.
- Consider the time it takes for a customer to pay for their meal. Currently this is not in our model despite being a very important factor in service times. This could make our results more accurate to the real-world scenario.

8 Conclusion

Murphy's Pub has been a staple for many Seattle locals and we have taken great strides in improving the staffing cost for the pub owners without sacrificing the quality of service that many long-standing patrons have grown accustomed to. Based on our results, 2-3 servers are needed on average to maintain high efficiency and customer satisfaction. From observing which days of the week benefit most in improving satisfaction rates when including an additional server (mostly jumping from 2 to 3 servers) we were able to make inferences on when a third server may be most beneficial.

By using the results obtained from our study, the owners of Murphy's Pub can minimize wasted labor by observing which days risk the least satisfaction changes when scheduling less servers, and experimenting with reduced labor amounts on given days.

9 Acknowledgement

We would like to thank our contact at Murphy's Pub, manager and owner Chelley Basset, our professor, Dr. Sara Billey, our reviewer and TA, Tony Zeng, for their collaboration in realizing this project. We would also like to thank the authors of [1] as their paper provided us with great inspiration.

References

[1] Jessica Fay Irena Chen and Melissa Stadt. "Increasing Efficiency for United Way's Free Tax Campaign". In: *SIAM Undergraduate Research Online* 11 (Mar. 2018). DOI: `10.1137/17S015768`.

[2] Mikhail Koroteev et al. "Optimization of Food Industry Production Using the Monte Carlo Simulation Method: A Case Study of a Meat Processing Plant". In: *Informatics* 9.1 (2022). ISSN: 2227-9709. DOI: `10.3390/informatics9010005`. URL: `https://www.mdpi.com/2227-9709/9/1/5`.

[3] Dmitry Sedov. "Restaurant closures during the COVID-19 pandemic: A descriptive analysis". In: *Economics Letters* 213 (2022), p. 110380. ISSN: 0165-1765. DOI: `https://doi.org/10.1016/j.econlet.2022.110380`. URL: `https://www.sciencedirect.com/science/article/pii/S0165176522000593`.

[4] Jeffrey B. Goldberg Wayne L. Winston. *Operations research: applications and algorithms*. Thomson/Brooks/Cole, 2004. ISBN: 9780534380588.

10 Appendix

10.1 Math behind model

We use the following attributes,

d : Numbered day of the week starting Sunday and ending Saturday, $d = 0, 1, 2, ..., 6$

N_d : Number of customers on day d

L_d : Multiset of ticket lifespans collected from all day d's observed

$\bar{L_d}$: Set of unique values in L_d

We defined satisfied and unsatisfied as follows. Let $\mathcal{C} = \{1, 2, 3, ..., N_d\} \subset \mathbb{N}$, $i \in \mathcal{C}$, and $j \in \mathbb{N}$, then we define R_{ij} as the time waited in queue to fulfill request j made by customer i, and $\Omega_i = \{R_{ij}\}_{j \in \mathbb{N}}$ denotes the set of all request wait times (if customer i made only 3 requests total, then $R_{ij} = 0$ for $j > 3$). Then we say customer i is satisfied if $\sup(\Omega_i) \leq 5$, or unsatisfied if $\sup(\Omega_i) > 5$. We define the satisfaction status of customer i as,

$$C_i := \begin{cases} 0, & \sup(\Omega_i) \leq 5 \\ 1, & \sup(\Omega_i) > 5 \end{cases}$$

and the total number of unsatisfied customers is given by,

$$U := \sum_{i=1}^{N_d} C_i$$

so the total number of satisfied customers is,

$$U_s := N_d - U.$$

The percentage of satisfied customers on day d is calculated as,

$$S_{U_s}(d) = \frac{U_n}{N_d}$$

and similarly for the unsatisfied percentage,

$$S_U(d) = \frac{U}{N_d}.$$

Since a customer's length of stay is probabilistic we used the following probability distribution to determine a customer's length of stay. Let $u \in \bar{L_d}$. The multiplicity of u, denoted $|u|$, is given by the number of instances of u in L_d. For example if on day d there were 5 tickets opened with lifespans 1,3,3,4,5, (1 ticket open for 1 minute, 2 for 3 minutes, etc) then $L_d = \{1, 3, 3, 4, 5\}$ and $\bar{L_d} = \{1, 3, 4, 5\}$. So the multiplicity of $u = 1$ is $|1| = 1$ and for $u = 3$, $|3| = 2$. Then the probability customer i stays for U minutes on day d is given by,

$$p(u) = \frac{|u|}{T_d}$$

Since there was no available data for service times, data for service times was collected on site on. Let Sr denote the multiset of service times over all days observed, and $\bar{Sr}$ the set of unique values. Then for $v \in \bar{Sr}$, the probability that a server takes v minutes to fulfill a request is given by,

$$ST := \frac{|v|}{|Sr|}$$

where $|Sr|$ is the cardinality of Sr (calculated by summing the multiplicity of each value in $\bar{Sr}$).

To generate the time until the next customer arrives in our simulation, we use the previously derived exponential distribution. However because the arrival rate increases throughout the day, we update the exponential distribution used with each hour passed throughout the day. This in turn increases the likelihood of quicker succession between customer parties during peak hours and then decreases the arrival rate during closing times.

To generate random numbers from the exponential distribution using a uniform number generator between 0 and 1, we must invert the previously derived Cumulative distribution function

$$F(x) = 1 - e^{-\lambda x}$$

$F(x)$ is the probability that a customer arrives within time x or less. If we invert the function so that for a given probability we output the exact time such that the probability describes arriving within that time or less, we can generate inter-arrival times using uniform random numbers between 0 and 1.

$$F^{-1}(\mu) = t = \frac{\ln(1-\mu)}{(-\lambda)}$$

Thus whenever a customer arrives we generate a value between 0 and 1, find the recorded lambda corresponding to the current time and day, then plug it into our function to generate a time until the next customer arrives.

We collected general data for servers to complete various requests such as taking a customer party's order and delivering their Drinks. To generate the time for a server to complete a party's requests, we randomly sample a pre-recorded time from our data.

We used the total time tickets were open to approximate the duration for which customers stay, and created a discrete probability distribution approximating the probability a party stays from 0-10 minutes, 10-20 minutes, ..., 190-200 minutes. We cut off tickets longer than 200 minutes.

10.2 Server Work Flow

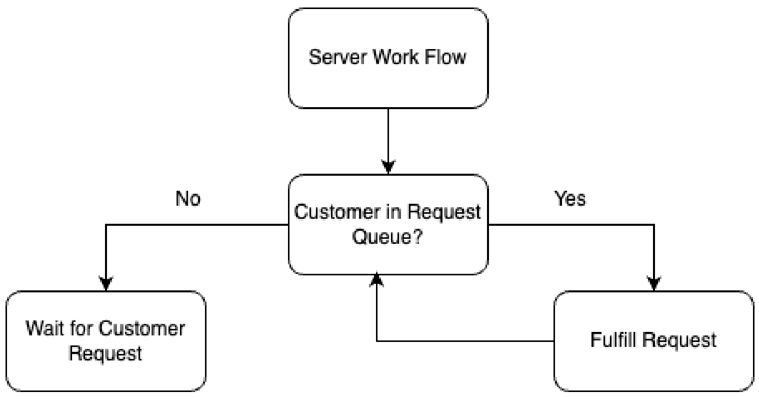


Figure 9: Server Workflow

MurphySim (4)

June 5, 2024

[166]:
```
import numpy as np
import math
import random as r
import queue as q
import pandas as pd
import matplotlib.pyplot as plt
```

[165]:
```
# indexed so that tickets[day][hour] is the list of␣
↪[avgTableSitters,avgBarSitters]
tickets = [
    [[3.4, 0.3], [4.9, 0.8], [5.7, 1.0], [6.3, 1.1], [4.5, 1.5], [5.4, 2.9], [6.
↪6, 3.7], [6.8, 2.7], [8.0, 2.8], [6.5, 2.6], [3.1, 2.8], [1.6, 2.2], [0.9, 1.
↪7], [0.3, 3.4]], #sunday (day=0)
    [[3.3, 3.7], [6.2, 2.5], [8.1, 4.2], [9.8, 4.0], [8.4, 3.5], [6.2, 3.7], [3.
↪8, 5.6], [0.6, 5.7], [0.1, 4.3]], #monday (day=1)
    [[4.0, 2.9], [4.6, 7.6], [11.4, 5.3], [14.1, 4.3], [8.0, 4.6], [2.6, 3.3],␣
↪[2.7, 4.2], [0.6, 2.8], [0.4, 2.8]], #etc.
    [[4.0, 0.8], [5.6, 1.2], [11.2, 2.5], [9.8, 3.1], [10.3, 5.0], [6.0, 4.9],␣
↪[5.3, 6.6], [2.2, 2.3], [0.5, 2.2]],
    [[4.1, 1.9], [5.8, 2.7], [10.3, 3.9], [11.1, 3.4], [9.0, 3.9], [6.1, 3.8],␣
↪[4.2, 5.2], [0.8, 5.2], [0.1, 3.9]],
    [[5.2, 1.7], [5.2, 1.6], [3.5, 2.3], [5.7, 3.8], [9.4, 10.7], [13.0, 5.9],␣
↪[15.3, 7.8], [11.4, 6.1], [10.1, 9.4], [7.0, 13.4], [3.8, 12.7], [1.9, 7.3],␣
↪[1.2, 2.5], [0.5, 2.4]],
    [[2.0, 0.9], [3.6, 1.5], [4.4, 1.9], [5.3, 2.7], [4.3, 2.2], [6.6, 3.1], [9.
↪1, 4.5], [11.3, 3.2], [11.2, 2.6], [12.1, 4.7], [9.6, 5.4], [7.2, 12.1], [3.
↪8, 10.1], [1.5, 6.1], [1.0, 3.5], [0.8, 2.1]]
]

def getMX(day):
    return 60 * len(tickets[day]) # convert to minutes

def getTableSitterProb(day, hour):
    numTableSittersAtHour = tickets[day][hour][0]
    numBarSittersAtHour = tickets[day][hour][1]
    return numTableSittersAtHour / (numTableSittersAtHour + numBarSittersAtHour)
```

```
def getAvgPeopleAtHour(day, hour):
    return tickets[day][hour][0] + tickets[day][hour][1]

def getIT(day, hour):
    averagePeoplePerMinute = getAvgPeopleAtHour(day,hour) / 60
    minUntilNextArrival = np.random.exponential(scale= (1 /␣
↪averagePeoplePerMinute), size=None)
    return minUntilNextArrival #60/arrivalRateAtHour # converts to avg minutes␣
↪til customer arrives
```

[198]:

```
#takes a CDF as input
class discreteDist:
    def __init__(self, cdf):
        self.cdf=cdf

    def generate(self):

        value=self.cdf.iloc[0]["x"]
        rand=np.random.randint(100)

        #print(rand)
        for index in range(len(self.cdf)):
            #print(self.cdf.iloc[index]["P(X<=x)"])

            test=self.cdf.iloc[index]["P(X<=x)"]
            if (rand>test):
                value=self.cdf.iloc[index+1]["x"]


        return value

# calculating estimated departure time cdfs
tableDCDF={
    "x": [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150,␣
↪160, 170, 180, 190, 200],
    "P(X<=x)": [6.32, 9.86, 16.06, 27.58, 41.12, 52.56, 62.22, 69.02, 74.33, 78.
↪17, 81.24, 84.07, 86.65, 88.98, 90.57, 92.09, 93.31, 94.21, 95.12, 100]
}
barDCDF={
    "x": [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150,␣
↪160, 170, 180, 190, 200],
    "P(X<=x)": [70.52, 72.1, 73.99, 75.74, 77.72, 79.72, 81.56, 83.71, 85.06,␣
↪86.61, 88.2, 89.35, 90.62, 91.58, 92.32, 93.14, 93.87, 94.46, 95.03, 100]
}
```

```
############## FIGURE THIS OUT LAST (we can do by trial and error and comparing␣
↪results {using inf servers} with real data)
RTCDF={
    "x": [35],
    "P(X<=x)": [100]
}

############## WE NEED TO USE REAL DATA FOR SERVICE TIME
STCDF={
    "x": [0.17, 0.22, 0.33, 0.33, 0.33, 0.4, 0.5, 0.5, 0.5, 0.6, 0.78, 0.87, 0.
↪93, 1.1, 1.3, 1.75, 2.0, 3.0, 3.57, 3.63, 4.0, 4.0, 4.33, 10.0, 20.0],
    "P(X<=x)": [4.0, 8.0, 12.0, 16.0, 20.0, 24.0, 28.0, 32.0, 36.0, 40.0, 44.0,␣
↪48.0, 52.0, 56.0, 60.0, 64.0, 68.0, 72.0, 76.0, 80.0, 84.0, 88.0, 92.0, 96.
↪0, 100]
}

tableDCDF=pd.DataFrame(tableDCDF)
barDCDF=pd.DataFrame(barDCDF)
STCDF=pd.DataFrame(STCDF)
RTCDF=pd.DataFrame(RTCDF)
```

[168]:
```
# Represents a time-based priority queue
class TimeQueue(q.PriorityQueue):
    def __init__(self):
        super().__init__()

    # Reads the "time" of the first element in the queue,
    # without modifying the queue.
    # If empty, returns infinity.
    def T(self):
        if self.empty():
            return np.inf
        first = self.get()
        super().put(first)
        return first[0]

    # Adds "data" to the queue with priority "time"
    def put(self, time, data):
        super().put((time, data))

    # Removes and returns the data of the highest-priority element
    def getData(self):
        return self.get()[1]
```

[169]:
```
# Class representing a customer
class Customer():
    #
    def __init__(self, ID):
        self.ID=ID

        self.leaveProb=0 #set value (0 bc people arent leaving if no tables are␣
↪available)

        self.estDepartureTime=0 #init (we dont know until the program starts)
        self.nextRequestTime=0 #init
        self.tablesitter=None #init

        # Dictionary for logging times of events
        self.time_log = {
            "gen":[], # time    of generation
            "req":[], # time(s) of request (starting)
            "fin":[], # time(s) of finishing (request)
            "dep":[], # time(?) of departure
            "blk":[], # time(?) of balking (if tablesitter and no empty tables)
            "wgt":[]  # time customer's request(s) spend waiting in queue (will␣
↪be negative if the request was not finished before time up)
        }

    def isTableSitter(self, prob):
        if np.random.random() <= prob:
            return True
        return False

    # Boolean, true if customer is table sitter, no tables, and decides to leave
    def isLeaving(self):
        if r.random() < self.leaveProb:
            return True
        return False

    # determines whether customer has time to order something else
    def hasFurtherRequest(self):
        if self.nextRequestTime < self.estDepartureTime:
            return True
        return False

    # setting up how to compare customers (for queue priority clashes)
    def __lt__(self, other):
        return self.ID < other.ID

    def __le__(self, other):
        return self.ID <= other.ID

    def __eq__(self, other):
        return self.ID == other.ID
    def __ge__(self, other):
        return self.ID > other.ID

    def __gt__(self, other):
        return self.ID >= other.ID

    def __ne__(self, other):
        return self.ID != other.ID
```

[170]:
```
# Runs the simulation until MX time
# ST is the total number of servers
# TT is the total number of tables
def simulation(S = 2, T = 16, day = 0):
    # INITIALIZE VARIABLES
    MX = getMX(day)
    AT = getIT(day, hour=0)    # When next customer arrives
    R = TimeQueue() # When requests occur
    F = TimeQueue() # When requests finish
    D = TimeQueue() # When customers depart
    WL = q.Queue()    # Custormers with unserved requests (FIFO)
    TableWL = q.Queue() # Tablesitters without a table
    S = S # Number of availble servers
    T = T # Number of availble tables

    all_customers = [] # list of all customers

    ID=0

    events=[]
    events.append(str(S)+" servers, "+str(T)+" tables")


    TM = min(AT, R.T(), F.T(), D.T()) #init set of time
    # Loop to run until we've simulated long enough
    while TM < MX:

        # Generate Customer
        if TM == AT:
            hour = math.floor(TM/60)
            c = Customer(ID=ID) # Generate new customer
            c.tablesitter = c.isTableSitter(getTableSitterProb(day, hour)) #␣
↪determine whether they sit at a table or not
            c.time_log['gen'].append(TM) # Log generation time
            all_customers.append(c)
            if c.tablesitter: #generate different departure time based on␣
↪whether they are at a table or not
```

```
                c.estDepartureTime=TM + discreteDist(tableDCDF).generate()
            else:
                c.estDepartureTime=TM + discreteDist(barDCDF).generate()
            events.append("Customer "+str(c.ID)+" arrived at TM="+str(TM)+",␣
↪and plans on leaving at "+str(c.estDepartureTime))
            AT += getIT(day, hour) # Set time of next new customer
            if c.tablesitter and T==0:
                if c.isLeaving():
                    events.append("Customer "+str(c.ID)+" is a tablesitter and␣
↪left because there's no tables")
                    c.time_log['blk'].append(TM)
                    pass   # Unsatisfied customer
                else:
                    events.append("Customer "+str(c.ID)+" is a tablesitter and␣
↪waits in the table waitlist")
                    TableWL.put(c) # Add to waitlist
            else:
                if c.tablesitter:
                    events.append("Customer "+str(c.ID)+" is a tablesitter and␣
↪sits at a table")
                    T -= 1 # Fill table
                    events.append("# of tables becomes "+str(T))
                else:
                    events.append("Customer "+str(c.ID)+" is not a tablesitter␣
↪and enters the pub")

                R.put(TM, c) # Add to queue

            ID+=1

        # Get new Request
        elif TM == R.T():
            target = R.getData() # Pop from Queue
            target.time_log['req'].append(TM) # Log request time
            events.append("Customer "+str(target.ID)+" wants to make a request␣
↪at TM="+str(TM))
            if S > 0:
                events.append("Customer "+str(target.ID)+" request is being␣
↪served at TM="+str(TM))
                S -= 1 # Customer served
                events.append("# of servers becomes "+str(S))

                generatedFT=discreteDist(STCDF).generate()
                events.append("Customer "+str(target.ID)+" will have their␣
↪request finished at TM="+str(TM+generatedFT))
```

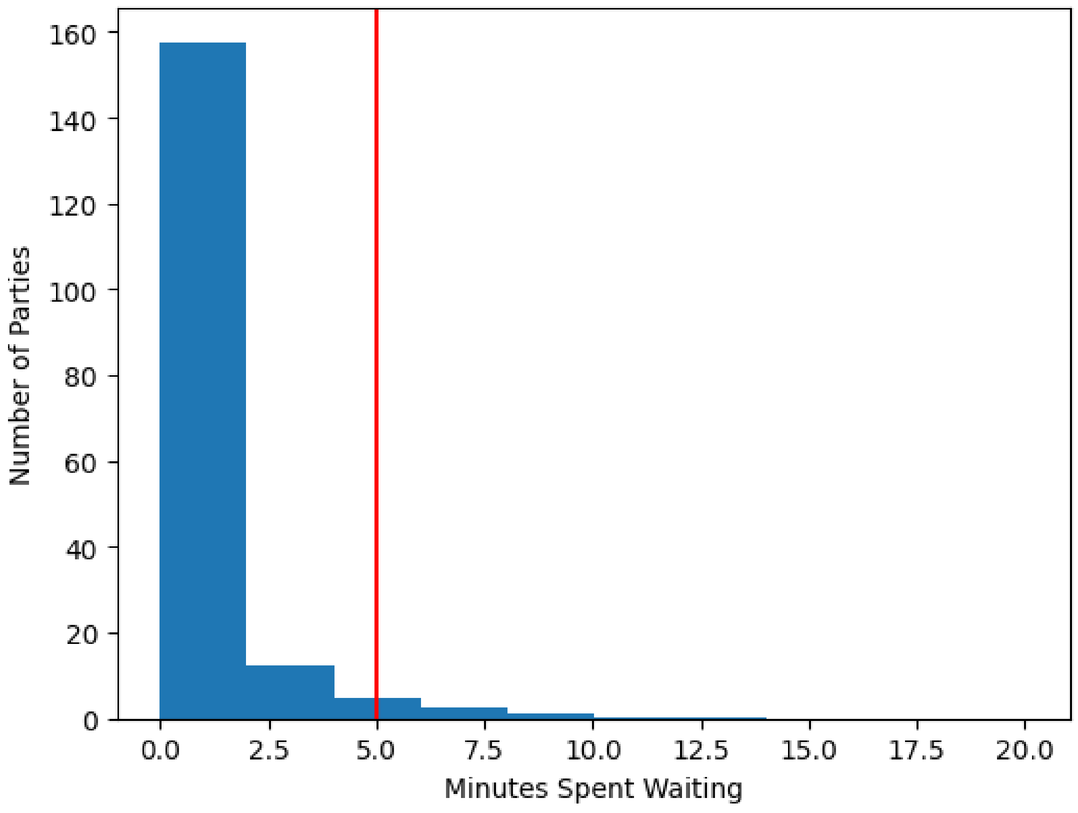
```
                F.put(TM + generatedFT, target) # Put a service finish time in␣
↪the queue
            else:
                events.append("Customer "+str(target.ID)+" request is put on a␣
↪waitlist")
                WL.put(target) # Put on WL

        # Request Finishes
        elif TM == F.T():
            target = F.getData() # Pop from Queue
            target.time_log['fin'].append(TM) # Log finish time
            events.append("Customer "+str(target.ID)+" request is finished at␣
↪TM="+str(TM))
            target.nextRequestTime=TM + discreteDist(RTCDF).generate()
            if target.hasFurtherRequest():
                events.append("Customer "+str(target.ID)+" will make a new␣
↪request at TM="+str(target.nextRequestTime))
                R.put(target.nextRequestTime, target) # Add to queue
            else:
                if TM > target.estDepartureTime:
                    events.append("Customer "+str(target.ID)+" has no more␣
↪requests and will depart at TM="+str(TM))
                    D.put(TM, target) # Start departure right now because they␣
↪overstayed
                else:
                    events.append("Customer "+str(target.ID)+" has no more␣
↪requests and will depart at TM="+str(target.estDepartureTime))
                    D.put(target.estDepartureTime, target) # Start departure␣
↪(maybe they chill for a bit before they leave)
            if WL.empty():
                S += 1 # Server becomes availible
                events.append("The waitlist is empty so the amount of servers␣
↪available becomes "+str(S))
            else:
                S += 1
                c = WL.get() # Pop from waitlist
                c.time_log['wgt'].append(TM - c.time_log['req'][-1])
                events.append("Customer "+str(c.ID)+" waited for WGT="+str(TM -␣
↪c.time_log['req'][-1]))
                events.append("Customer "+str(c.ID)+" request is taken off the␣
↪waitlist at TM="+str(TM))
                R.put(TM, c)

        # Customer Departs
        elif TM == D.T():
            target = D.getData() # Pop from Queue
            target.time_log['dep'].append(TM) # Log departure time

            events.append("Customer "+str(target.ID)+" departs at TM="+str(TM))

            if target.tablesitter:

                T += 1
                events.append("Customer "+str(target.ID)+" was a tablesitter so␣
↪the amount of available tables becomes "+str(T))
                if not TableWL.empty():
                    c = TableWL.get()


                    T -= 1 # Fill table
                    events.append("Customer "+str(c.ID)+" gets taken off the␣
↪table waitlist so the amount of available tables becomes "+str(T))
                    events.append("Customer "+str(c.ID)+" will make a request␣
↪at "+str(TM))

                    R.put(TM, c) # Add to queue

        # Should never get to this state, but might as well check
        else:
            print("Whoops!")
            return

        # Jump time to next event
        TM = min(AT, R.T(), F.T(), D.T())

    return all_customers, events
```

[222]:

```
NUMBER_OF_SERVERS = 3
DAY = 5
THRESHOLD = 5
NUMSIM = 100

sim_waits = []
for x in range(NUMSIM):
    waits = []
    sim = simulation(S = NUMBER_OF_SERVERS, T = 16, day = DAY)
    for customer in sim[0]:
        if customer.time_log['wgt']:
            waits.append(max(customer.time_log['wgt']))
        else:
            waits.append(0)
    sim_waits += waits
```

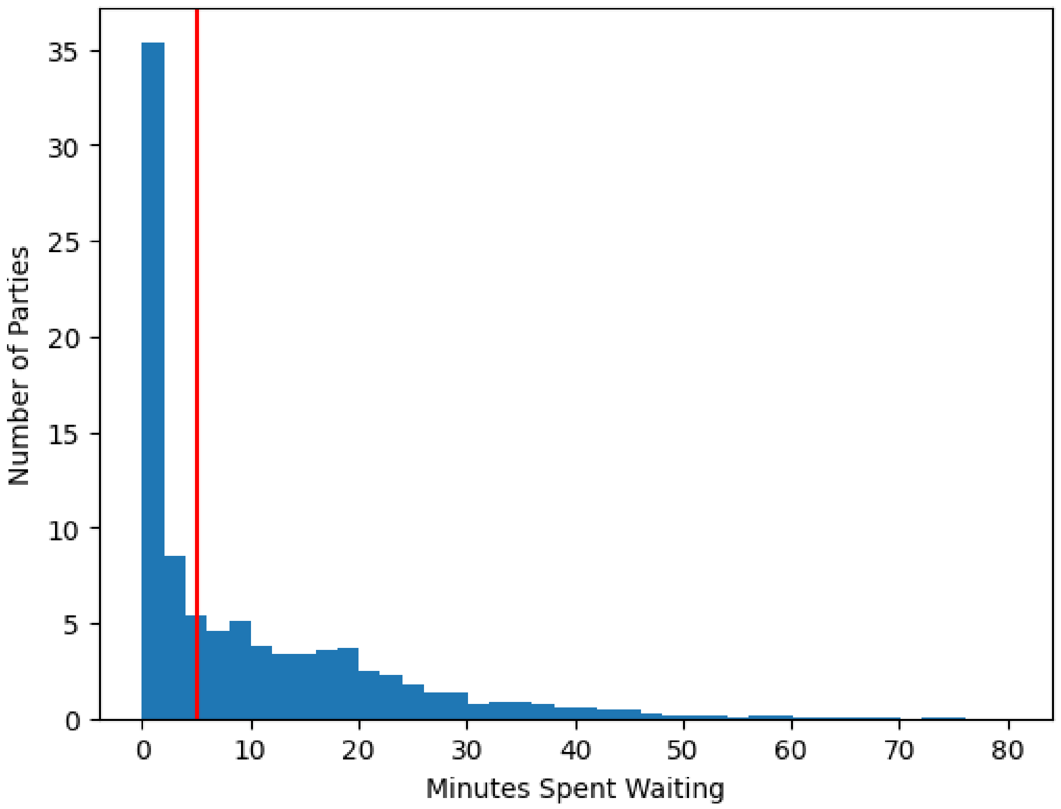
```
unsatisfied = len([time for time in sim_waits if time > THRESHOLD])
total_customers = len(sim_waits)
satisfied = total_customers - unsatisfied
plt.hist(sim_waits,bins=round(max(sim_waits)/2),weights=[1/
  ↪NUMSIM]*len(sim_waits))
plt.axvline(THRESHOLD, color="red")
plt.xlabel("Minutes Spent Waiting")
plt.ylabel("Number of Parties")
print(f'Number of servers           : {NUMBER_OF_SERVERS}')
print(f'Waited too long on waitlist : {unsatisfied}\t   {unsatisfied/
  ↪total_customers:.1%}')
print(f'Satisfied                   : {satisfied}\t     {satisfied/
  ↪total_customers:.1%}')
```

```
Number of servers           : 3
Waited too long on waitlist : 662          3.7%
Satisfied                   : 17251          96.3%
```

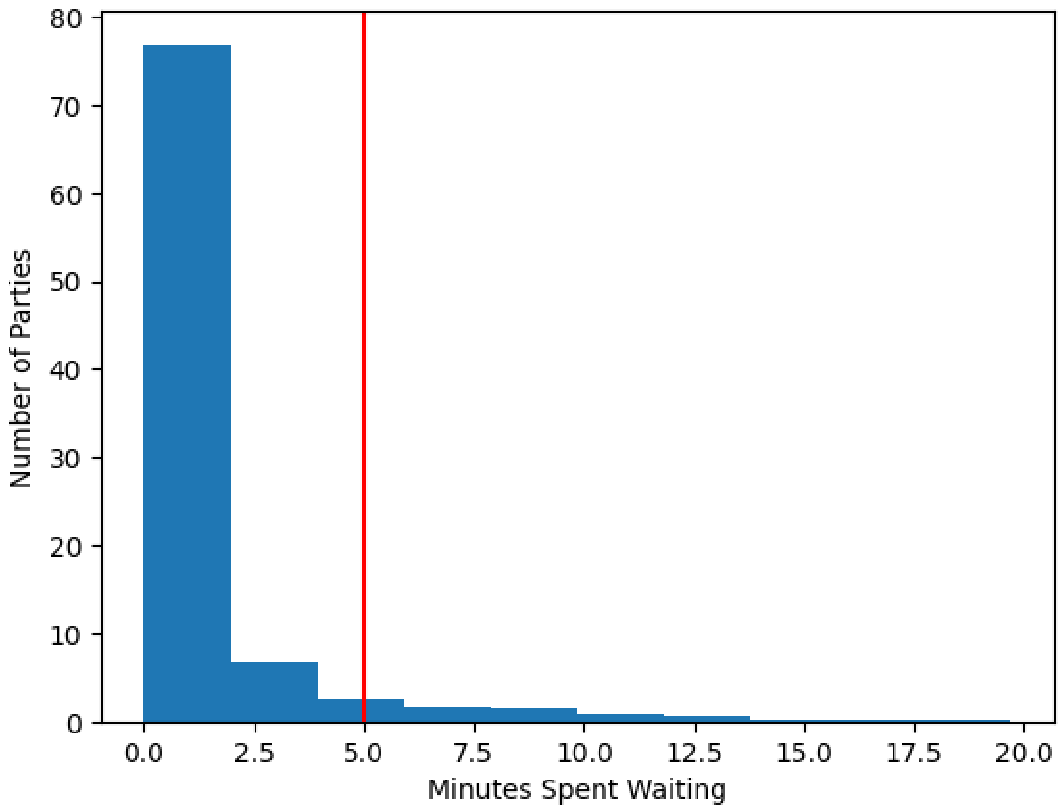


```
[228]: for dayNum in range(7):
           for numServer in range(1,4):
               NUMBER_OF_SERVERS = numServer
               DAY = dayNum
               THRESHOLD = 5
               NUMSIM = 100

               sim_waits = []
               for x in range(NUMSIM):
                   waits = []
                   sim = simulation(S = NUMBER_OF_SERVERS, T = 16, day = DAY)
                   for customer in sim[0]:
                       if customer.time_log['wgt']:
                           waits.append(max(customer.time_log['wgt']))
                       else:
                           waits.append(0)
                   sim_waits += waits


               unsatisfied = len([time for time in sim_waits if time > THRESHOLD])
               total_customers = len(sim_waits)
               satisfied = total_customers - unsatisfied
               plt.hist(sim_waits,bins=round(max(sim_waits)/2),weights=[1/
        ↪NUMSIM]*len(sim_waits))
               plt.axvline(THRESHOLD, color="red")
               plt.xlabel("Minutes Spent Waiting")
               plt.ylabel("Number of Parties")
               print(f'Day                          : {DAY}')
               print(f'Number of servers            : {NUMBER_OF_SERVERS}')
               print(f'Waited too long on waitlist : {unsatisfied}\t   {unsatisfied/
        ↪total_customers:.1%}')
               print(f'Satisfied                    : {satisfied}\t     {satisfied/
        ↪total_customers:.1%}')
               plt.show()
```

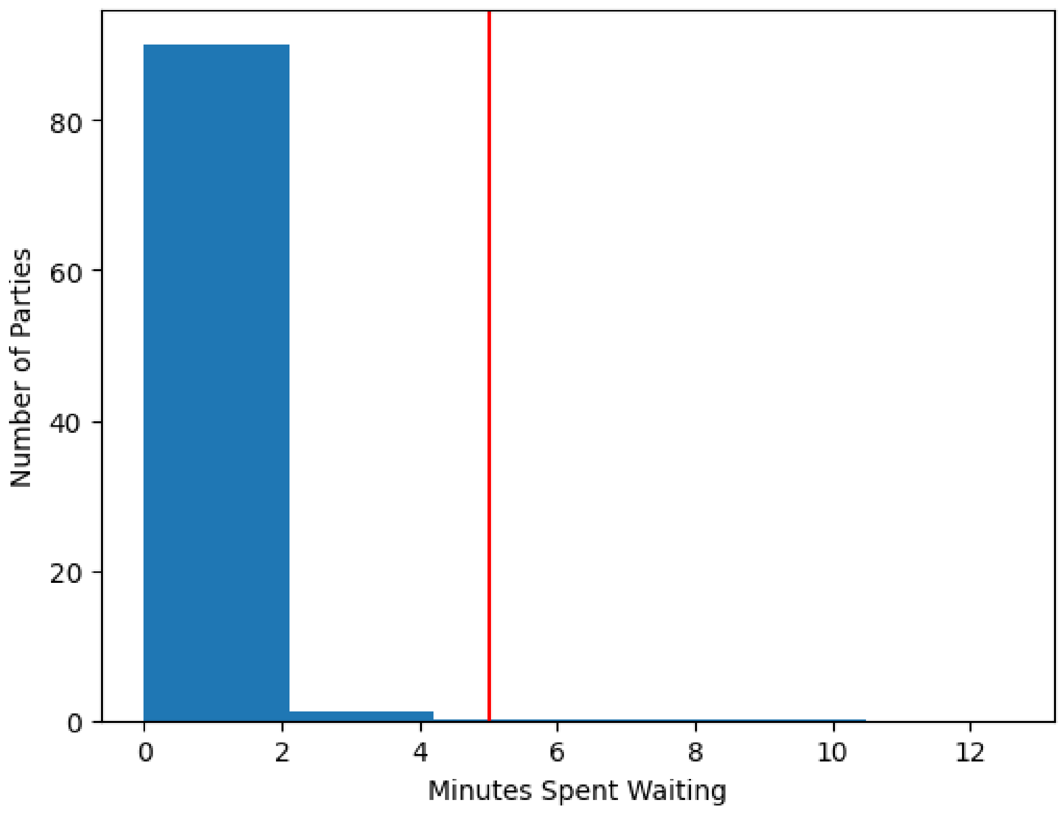
```
Day                         : 0
Number of servers           : 1
Waited too long on waitlist : 4652         49.9%
Satisfied                   : 4663           50.1%
```



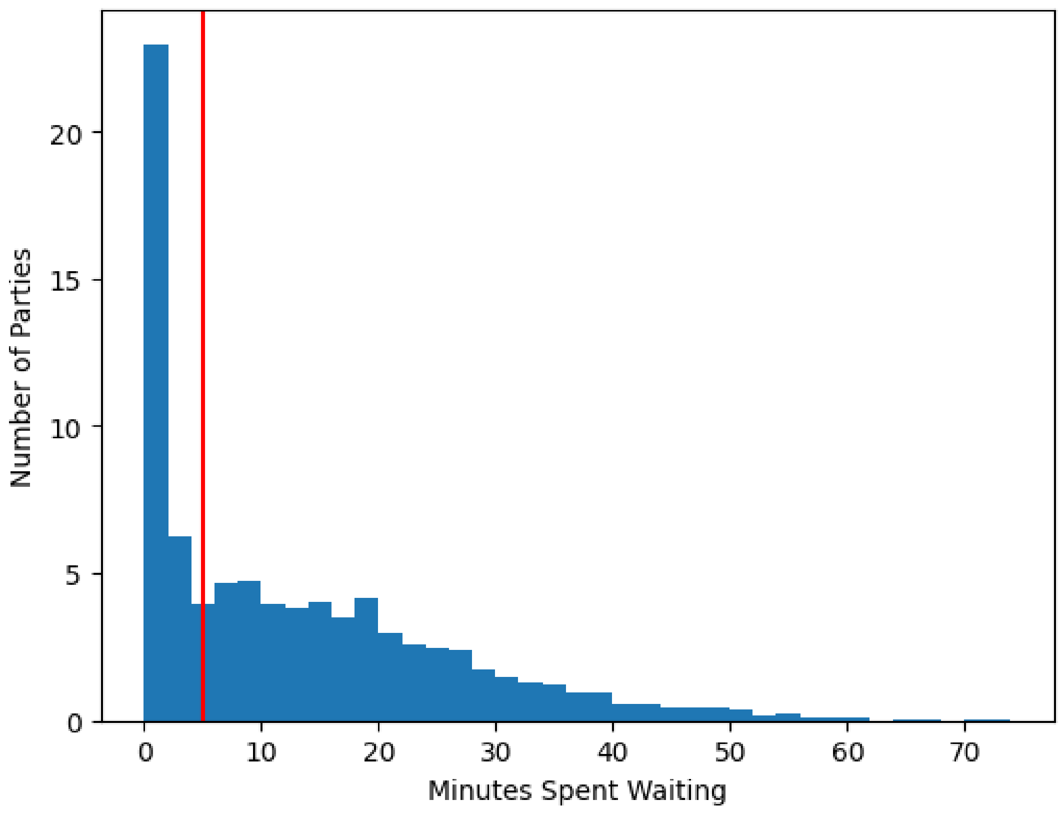
```
Day                         : 0
Number of servers           : 2
Waited too long on waitlist : 572          6.3%
Satisfied                   : 8528           93.7%
```



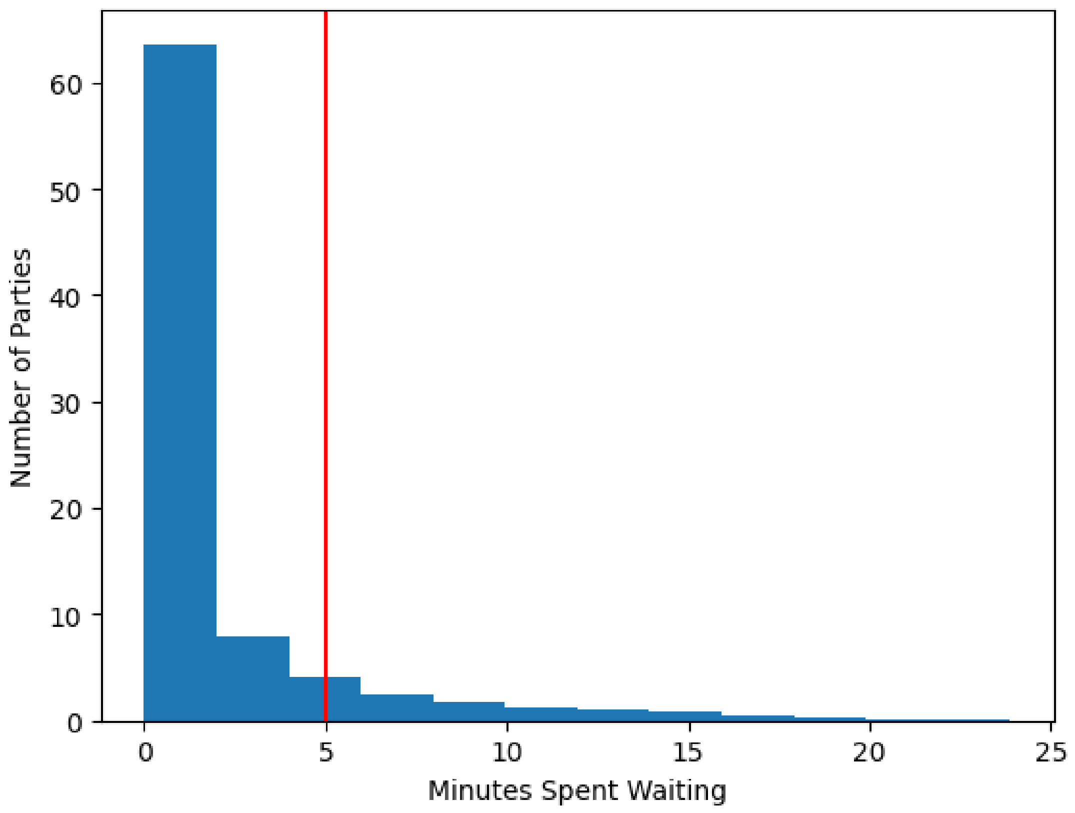
```
Day                          : 0
Number of servers            : 3
Waited too long on waitlist : 48           0.5%
Satisfied                    : 9144          99.5%
```



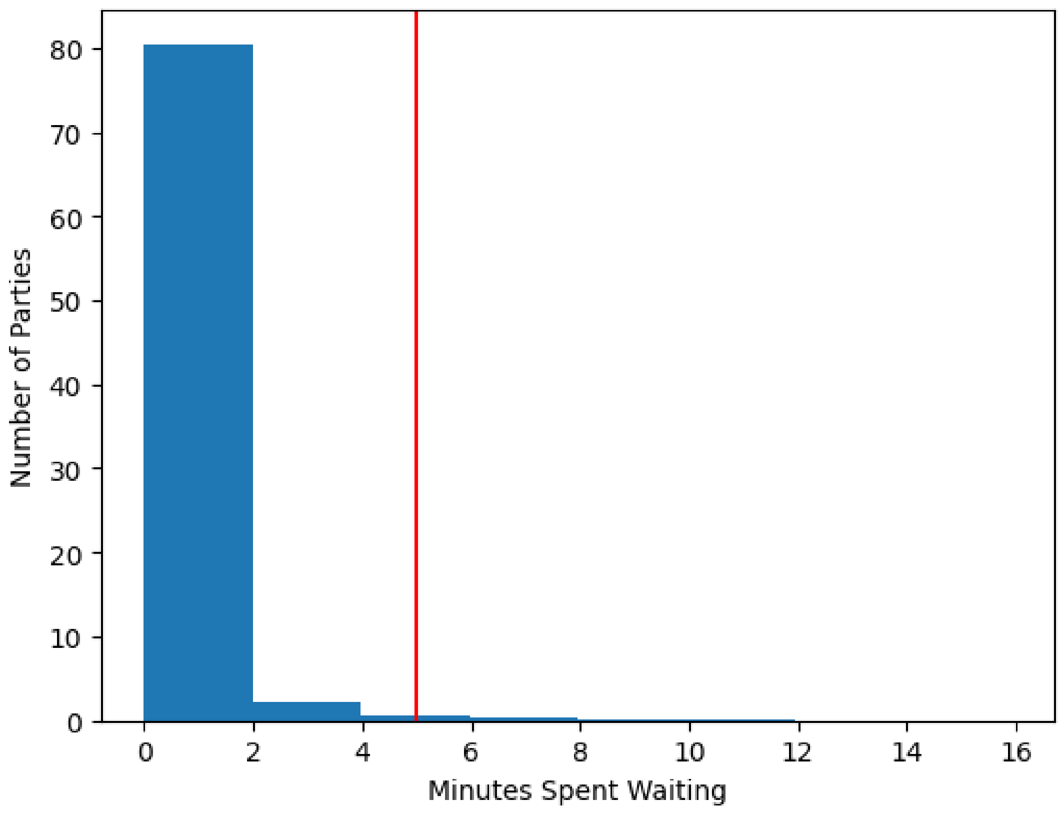
```
Day                          : 1
Number of servers            : 1
Waited too long on waitlist : 5273         62.8%
Satisfied                    : 3123          37.2%
```



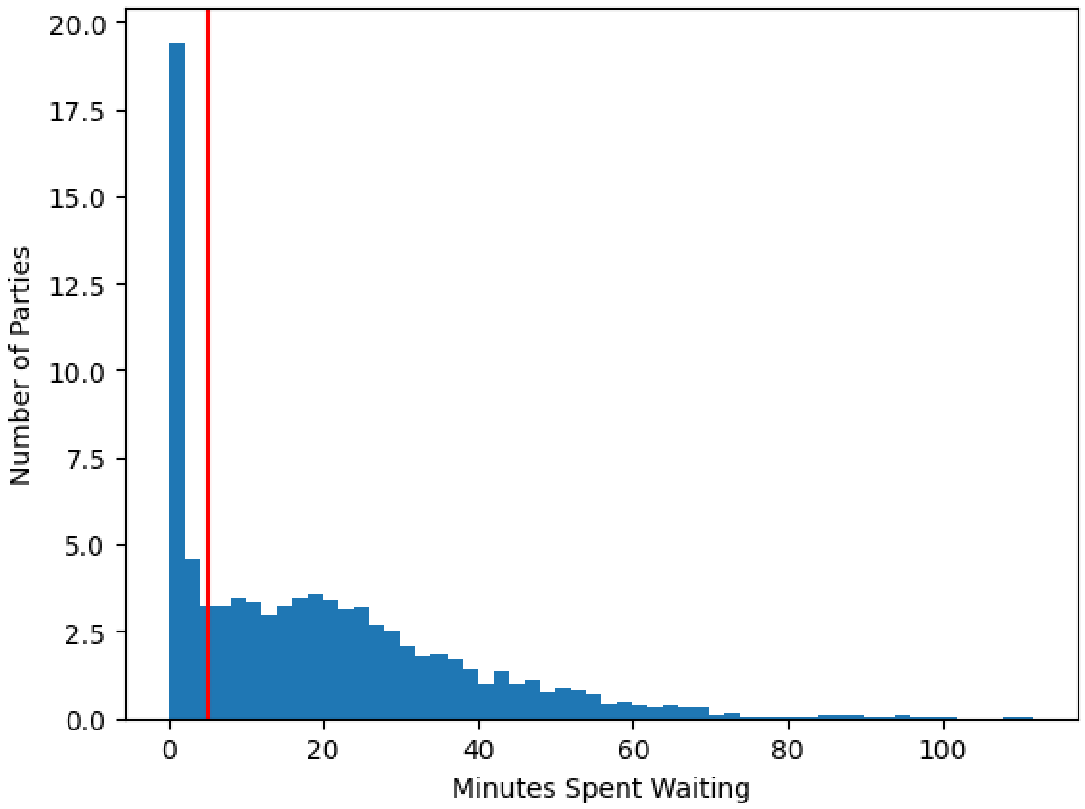
```
Day                          : 1
Number of servers            : 2
Waited too long on waitlist : 996         11.9%
Satisfied                    : 7397          88.1%
```



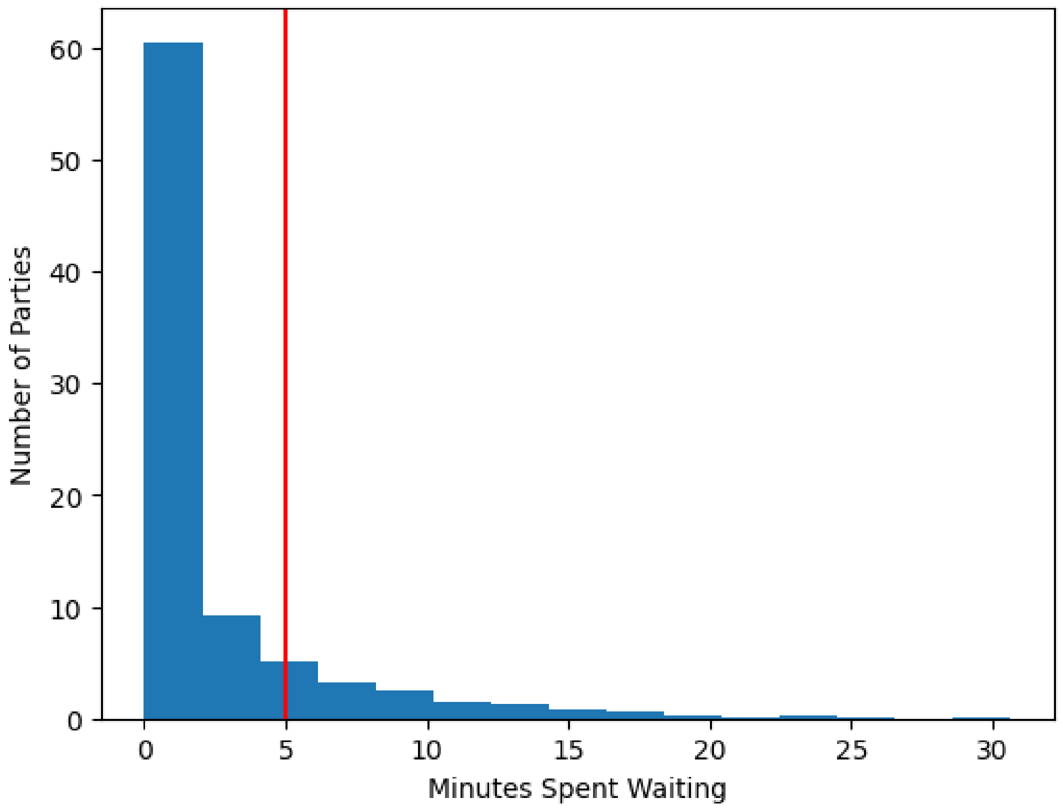
```
Day                         : 1
Number of servers           : 3
Waited too long on waitlist : 96          1.1%
Satisfied                   : 8301          98.9%
```



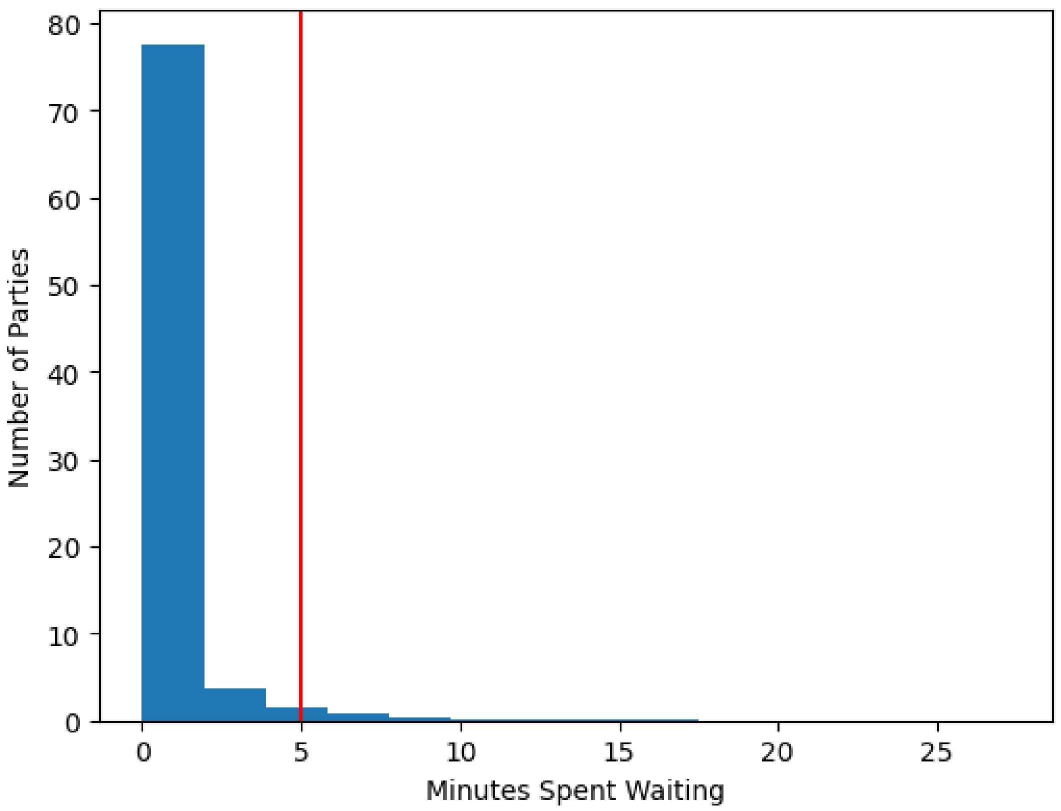
```
Day                          : 2
Number of servers            : 1
Waited too long on waitlist : 5972         70.0%
Satisfied                    : 2564           30.0%
```



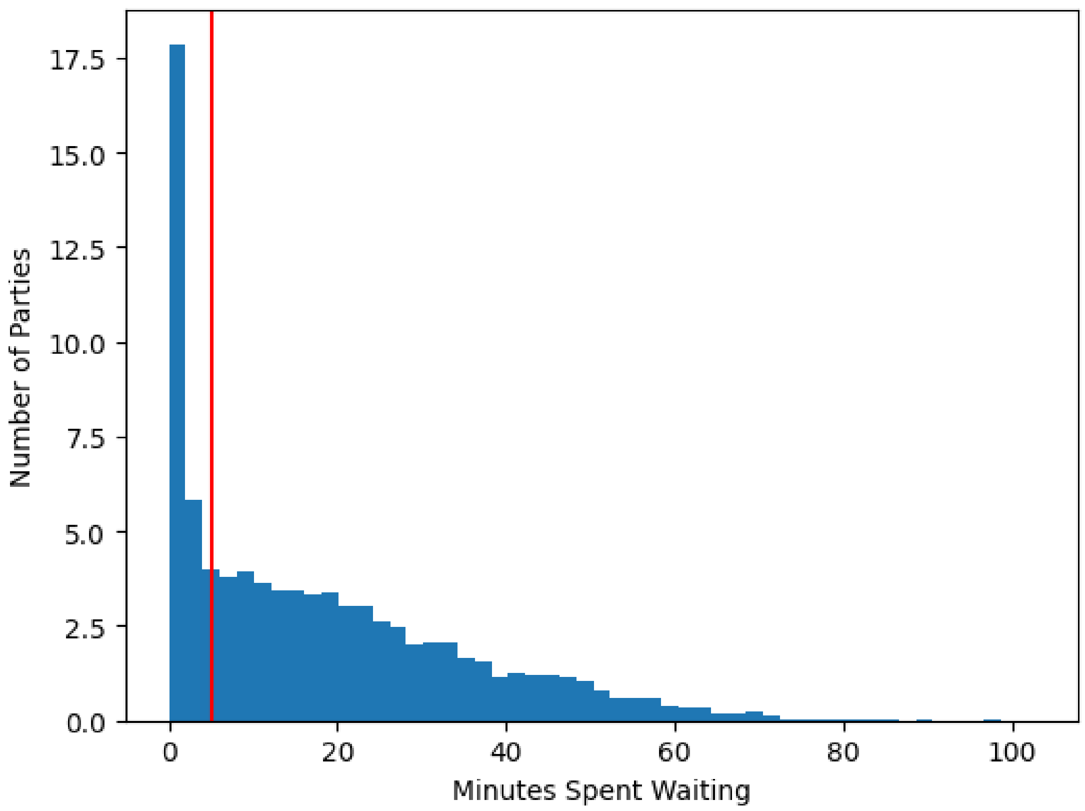
```
Day                          : 2
Number of servers            : 2
Waited too long on waitlist : 1326         15.5%
Satisfied                    : 7223          84.5%
```



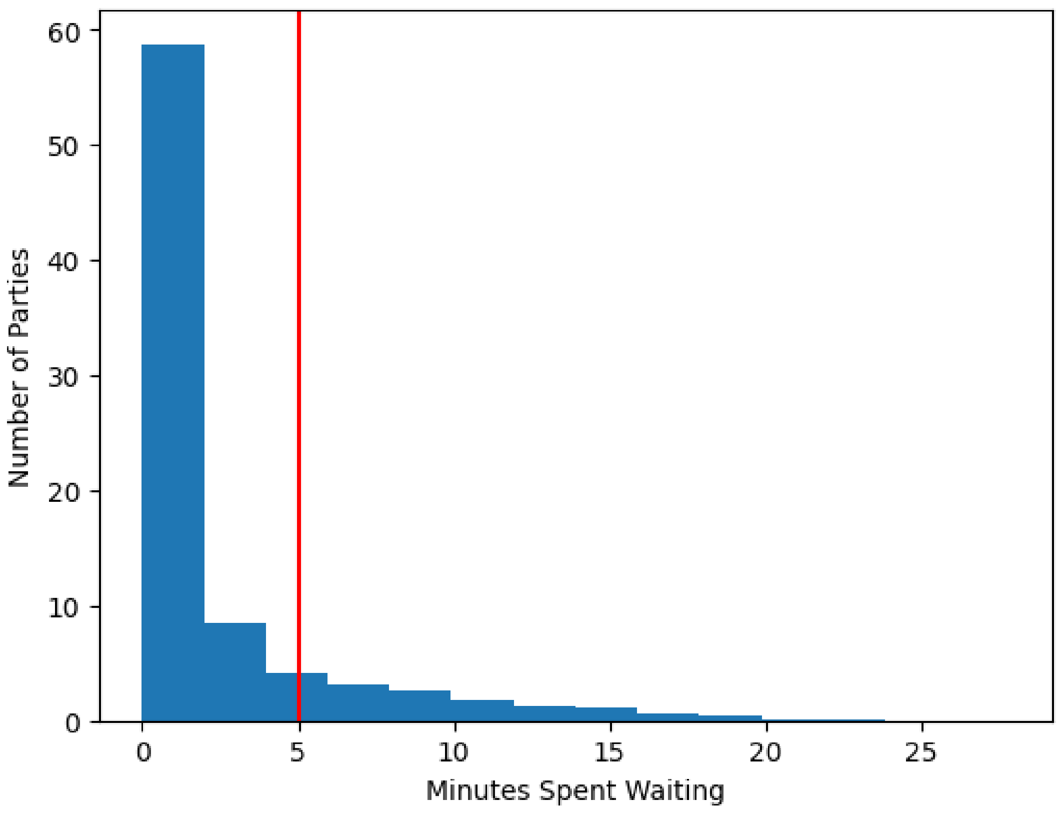
```
Day                          : 2
Number of servers            : 3
Waited too long on waitlist : 271         3.2%
Satisfied                    : 8227         96.8%
```



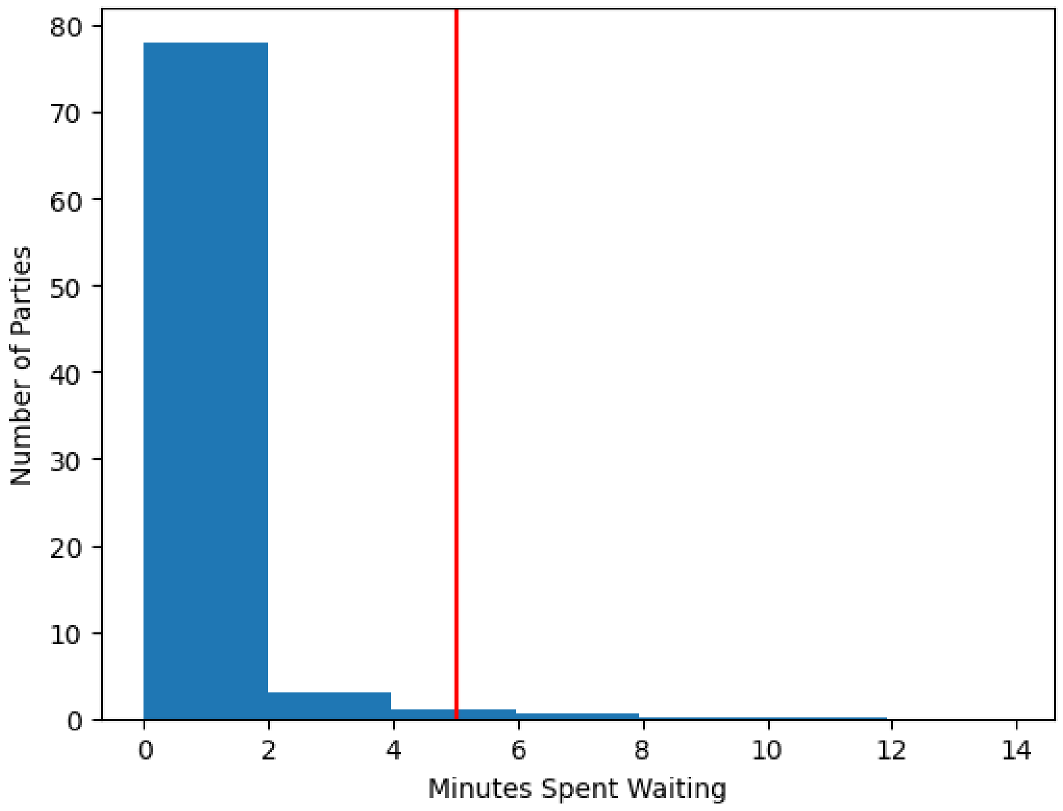
```
Day                          : 3
Number of servers            : 1
Waited too long on waitlist : 5948         69.9%
Satisfied                    : 2561           30.1%
```



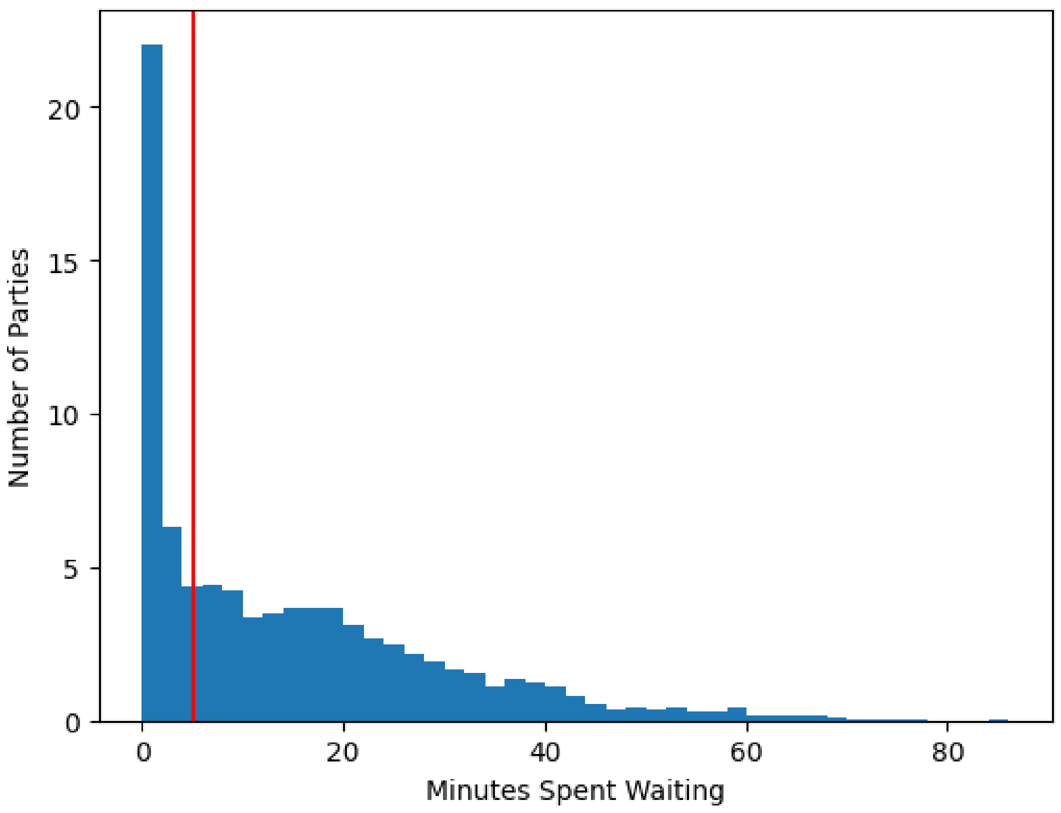
```
Day                          : 3
Number of servers            : 2
Waited too long on waitlist : 1324         16.0%
Satisfied                    : 6951          84.0%
```



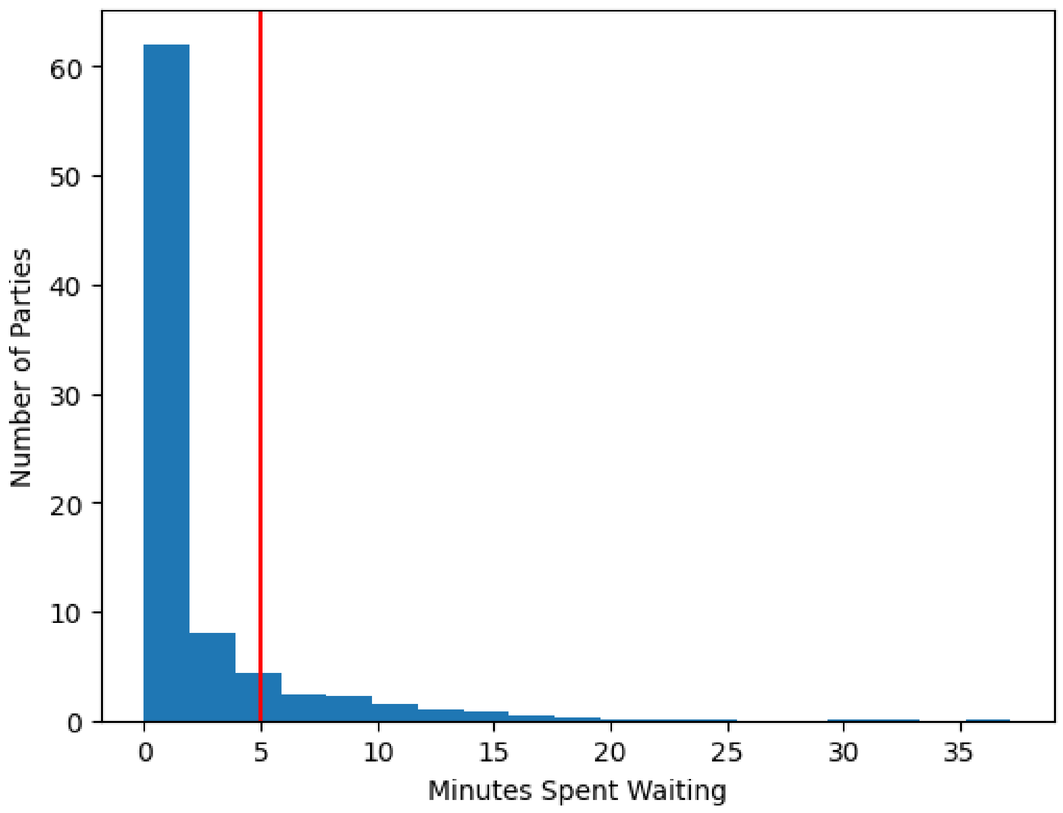
```
Day                         : 3
Number of servers           : 3
Waited too long on waitlist : 128          1.5%
Satisfied                   : 8162           98.5%
```



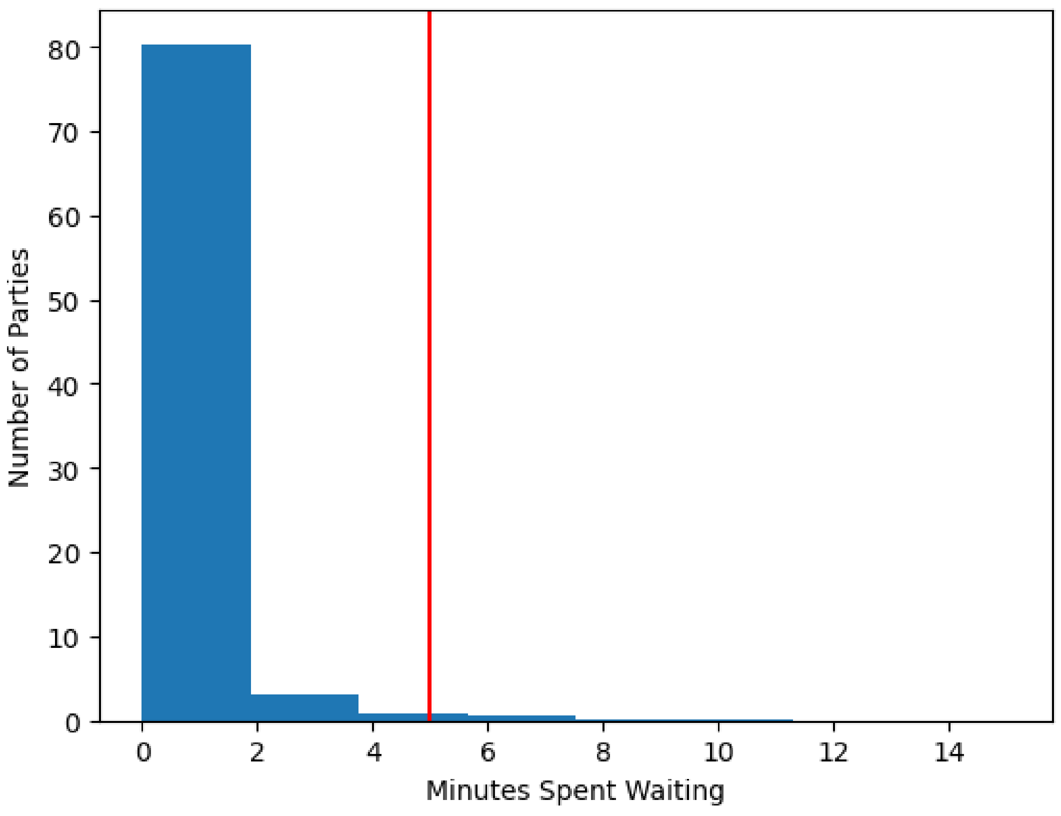
```
Day                          : 4
Number of servers            : 1
Waited too long on waitlist : 5408         63.8%
Satisfied                    : 3069          36.2%
```



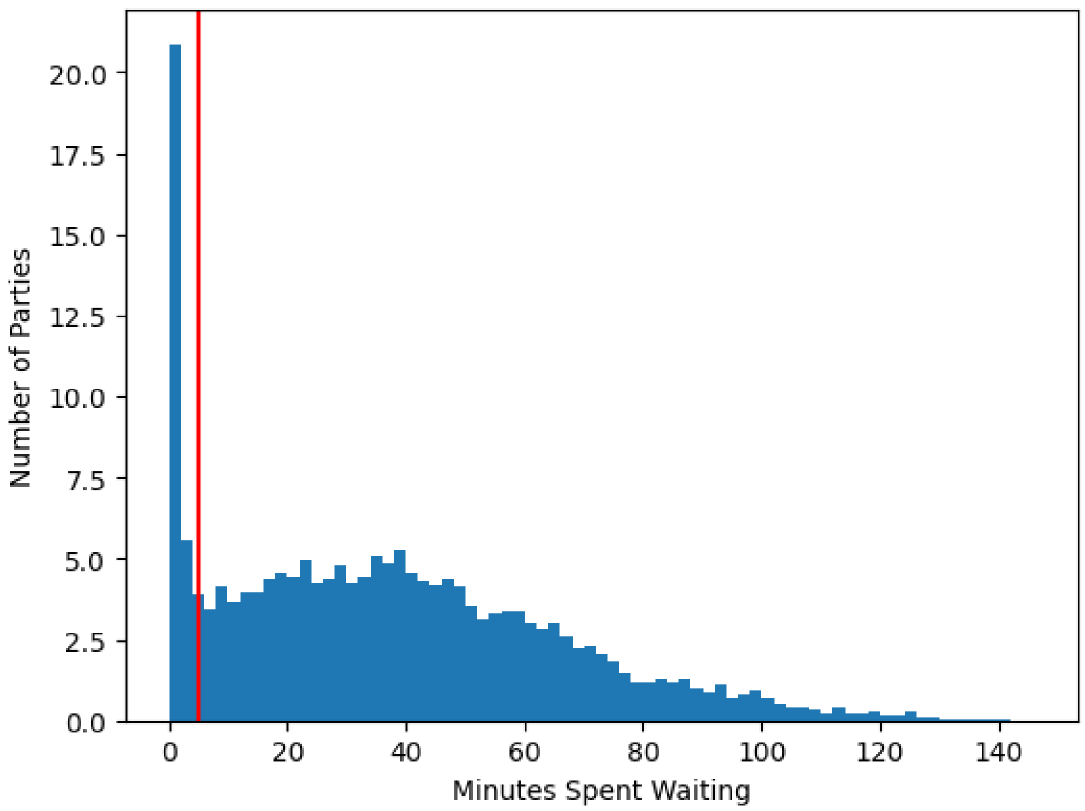
```
Day                          : 4
Number of servers            : 2
Waited too long on waitlist : 1085         13.0%
Satisfied                    : 7281           87.0%
```



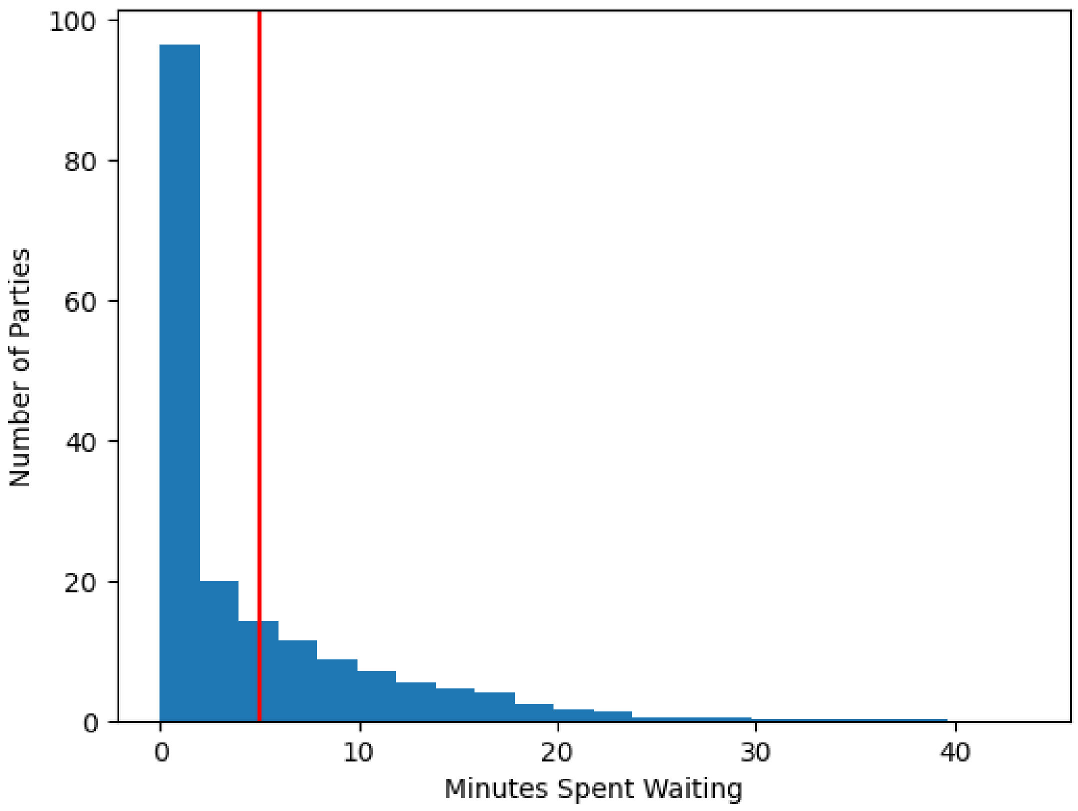
```
Day                          : 4
Number of servers            : 3
Waited too long on waitlist : 128          1.5%
Satisfied                    : 8406          98.5%
```



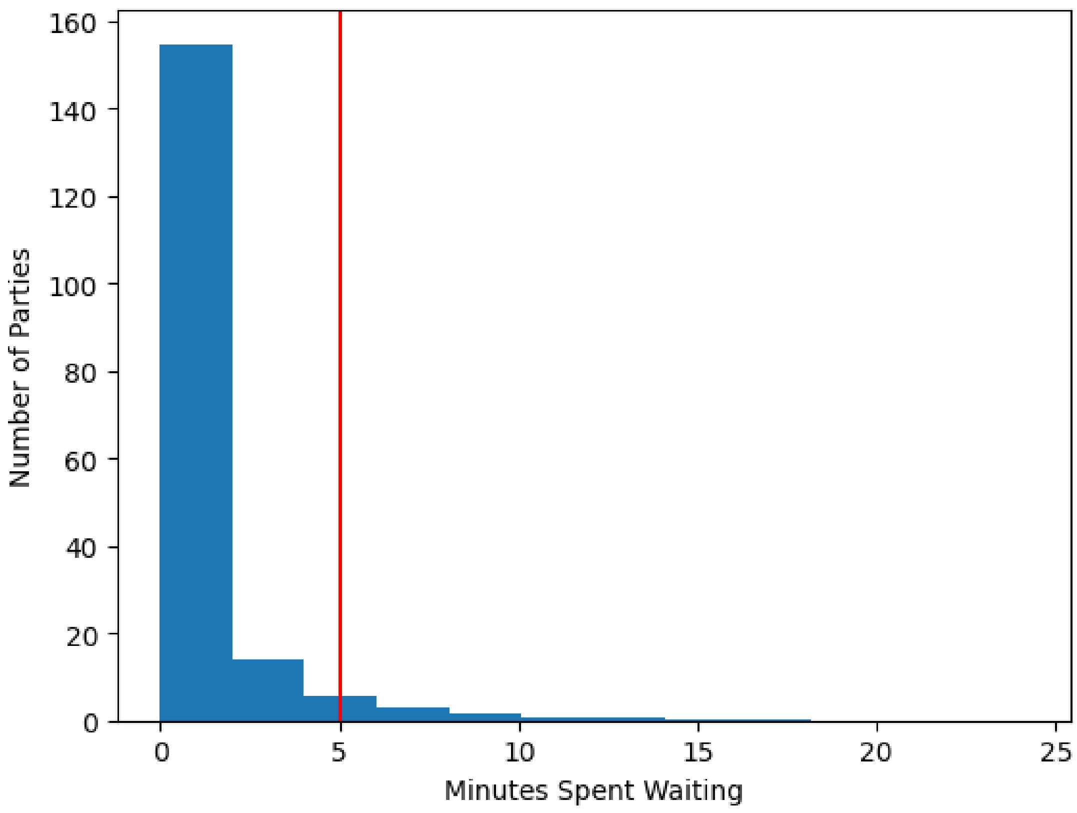
```
Day                          : 5
Number of servers            : 1
Waited too long on waitlist : 15221        84.3%
Satisfied                    : 2841           15.7%
```



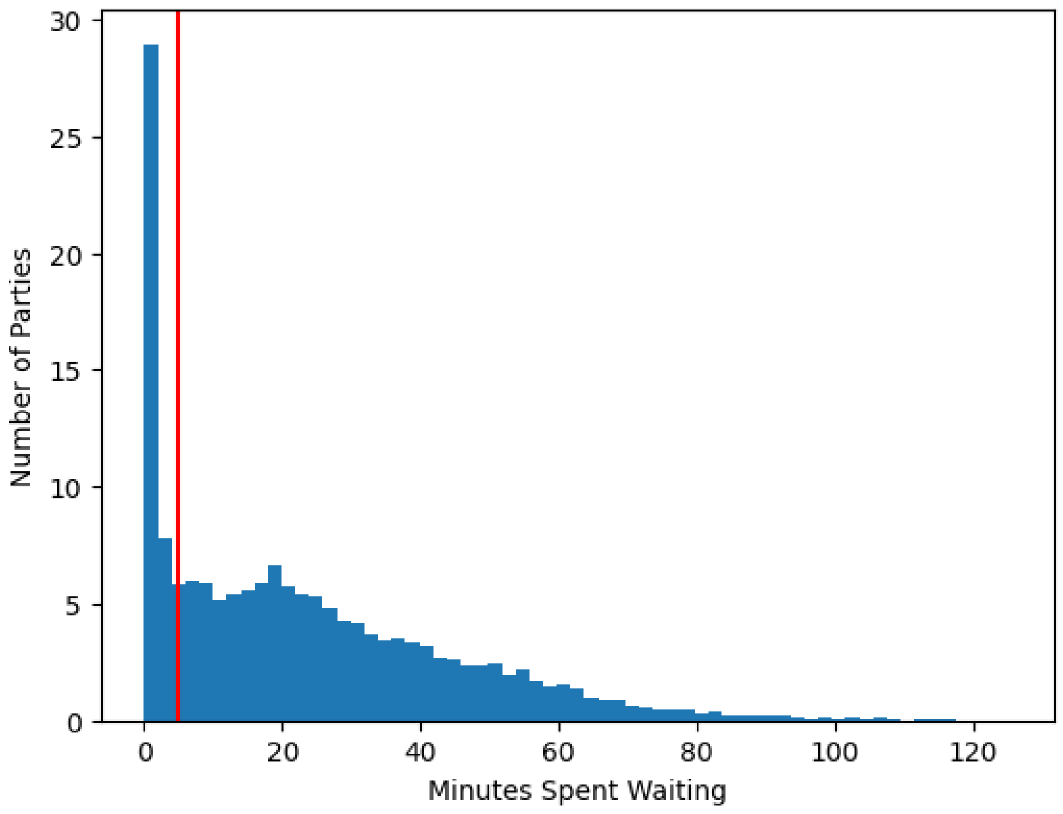
```
Day                          : 5
Number of servers            : 2
Waited too long on waitlist : 5513        30.7%
Satisfied                    : 12416        69.3%
```



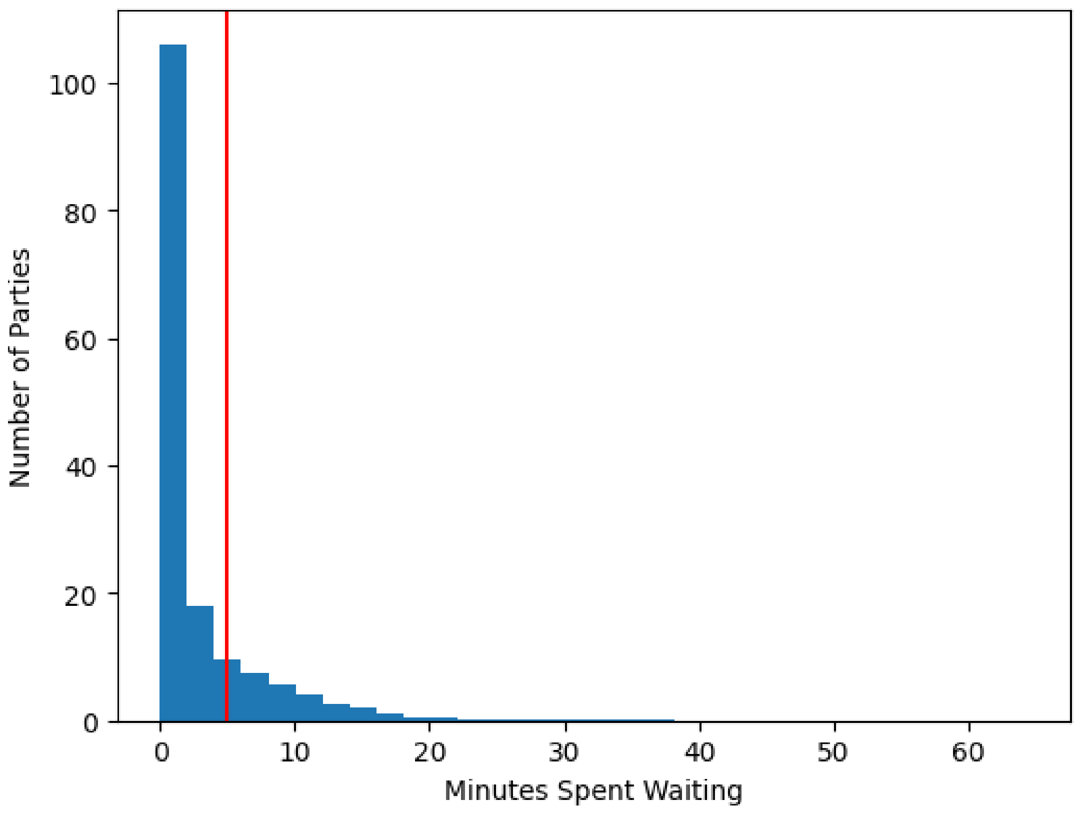
```
Day                          : 5
Number of servers            : 3
Waited too long on waitlist : 911          5.0%
Satisfied                    : 17178         95.0%
```



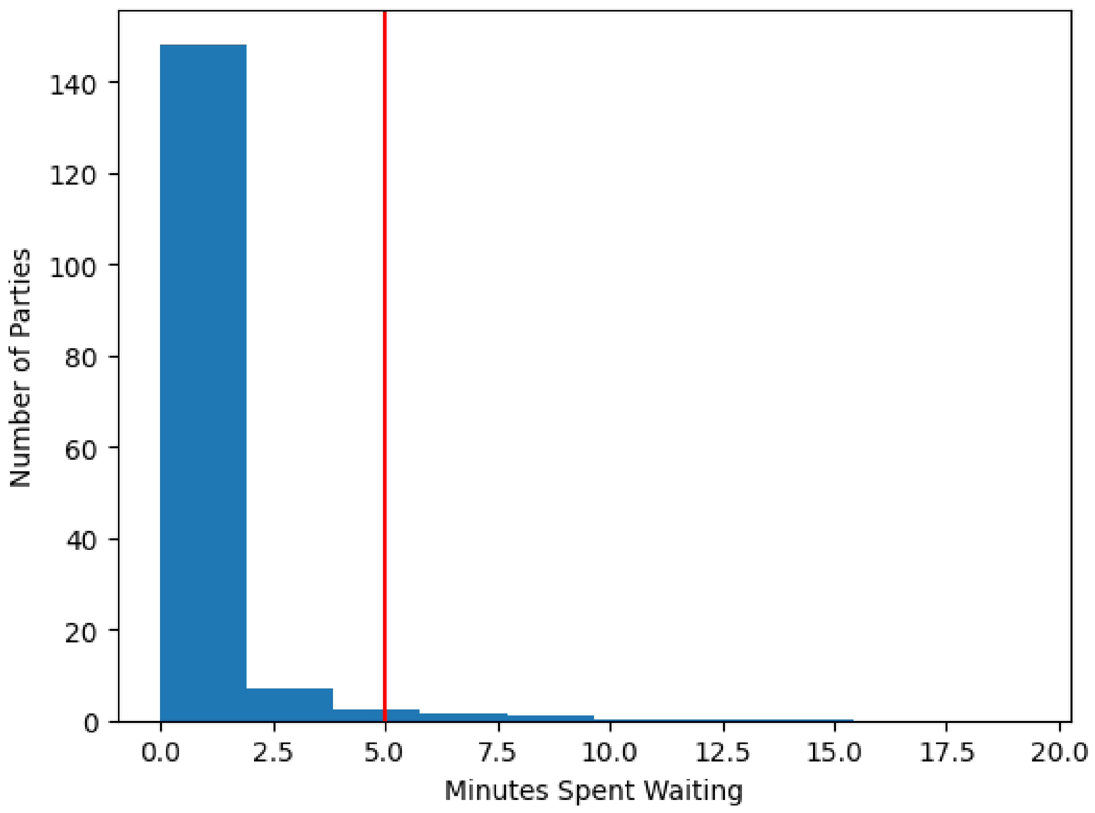
```
Day                          : 6
Number of servers            : 1
Waited too long on waitlist : 12116        75.3%
Satisfied                    : 3966          24.7%
```



```
Day                          : 6
Number of servers            : 2
Waited too long on waitlist : 3058          19.2%
Satisfied                    : 12893          80.8%
```



```
Day                         : 6
Number of servers           : 3
Waited too long on waitlist : 410          2.5%
Satisfied                   : 15721          97.5%
```



Number of Parties
Minutes Spent Waiting
0
20
40
60
80
100
120
140
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0