# Call Center Simulation

Matthew Tobino, Sean Pandolfo, Justin Pitera, Andy Pham

May 3, 2023

Intro to System Simulation and Modeling

Rowan University
Glassboro, NJ 08028

# Table of Contents

**Abstract**

According to Live Agent, a call center will average about 4,400 calls per month. This statistic is further broken down to 200 calls a day, 1,000 calls a week, and approximately 4,000 calls per month. The number of calls to a center and the amount of time spent with a caller will dramatically change depending on the industry the call center handles; however, it is important to make sure that the center has the proper staff with the proper training to handle the scenarios the customers bring efficiently and effectively. In this project, we modeled a call center to show the effects of having a low number of servers and a high number of servers. As one can imagine, having a low number of servers means a call center can only take so many calls before a large queue will form. With that large queue, it can be reasonably expected that some callers will simply hang up and try again another day after waiting an extended period of time or after hearing how long the wait time would be. With that in mind, we designed our project around the number of servers in a call center and showed how crucial it is to have the proper amount of staffing to run an efficient call center. The simulation will take in the number of servers the call center will have and then how many hours the center will be active. With these variables in mind, we will be able to simulate a call center based on our collected data and display the results of it.

**Background**

It is no secret that call centers provide an invaluable service to companies and corporations. Delegating the tasks of customer service, sales, and technical support gives companies massive amounts of leeway with their staff and how to better utilize their talents and expertise. After all, it is ill-advised to put computer scientists in sales. As of 2020, the global call center industry was valued at $340 billion and was projected to grow to $496 billion by 2027 (Estrellado, 2023). With such a high market value, companies must invest resources into developing their call centers to be up-to-date and efficient. However, the turnover rate for call centers remains around 30% to 45% (Branka, 2023). With such high turnover rates, companies are losing staff that could have easily helped solve some of their company's issues with service, sales, and technical support.

By simulating a call center, we can show companies the importance of maintaining their staff. Without the proper staff, many callers will hang up before they ever get a chance to be served. If the customer is not happy with the service provided, they can easily switch to another product/company that is better suited to their needs. The simulation we designed shows that many callers will drop out before they are serviced due to a lack of servers that can move the queue forward.

## Simulation

### *Importing The Data*

Python was a clear language of choice for our simulation because of its versatility, ease of use, and extensive libraries for data analysis. Python has become increasingly popular in the data science community, with many large companies such as Google and NASA using it for data analysis and machine learning. We believed that Python's powerful data analysis tools would enable us to extract insights from our call center data and create an accurate simulation.

To source data for our simulator, we utilized a dataset from Kaggle containing information on call centers. The dataset was in CSV format, which we could easily load into our Python script using the built-in CSV library. By analyzing the data, we were able to gain insights into call center performance.

Our Python script performed various data analysis tasks, such as calculating the average number of incoming calls and the average talk duration. We also used Python's random module to generate simulated call data based on the statistics we extracted from the dataset. By generating random data, we were able to simulate different scenarios and test the performance of our call center under different conditions.

Overall, we found that Python was an excellent choice for our call center simulator due to its data analysis capabilities and ease of use. By leveraging the power of Python's data analysis libraries, we were able to gain valuable insights into call center performance and improve the accuracy of our simulation.

### *Distribution & Data Used*

Poisson distributions are used in a few different places in the simulation. We use a random generator based on a Poisson distribution to get the number of customers calling per

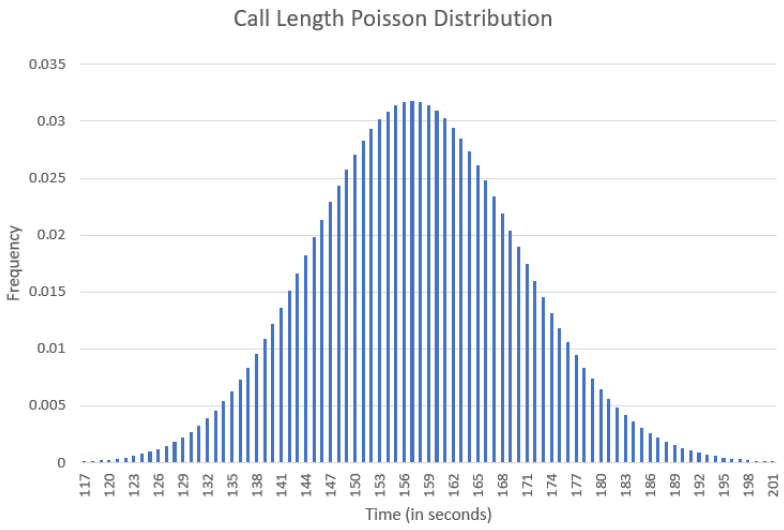
hour, as well as to generate the length of each call in seconds, and each customer's maximum waiting time.

The data we used from the dataset were *Calls per Day, Talk Duration (AVG)*, and *Waiting Time (AVG)*. We took the set of 550 values for each of these variables, calculated the average, and used the Python NumPy random.poisson function to generate random values.

***NumPy.random.poisson***

NumPy is a Python library that provides a litany of useful mathematical functions, and the *random.poisson()* function was perfect for the purposes of this project. The *NumPy.random.poisson()* function takes input of an average value, so for example, the average length of each call. This function will then output a random number - however this is not a completely random value - rather it is based on a Poisson distribution.

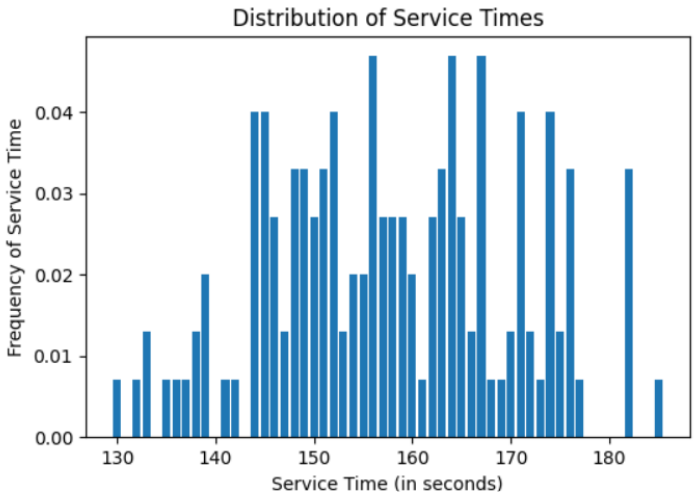
For reference, here is the graph of the poisson distribution of call lengths, generated in Excel:



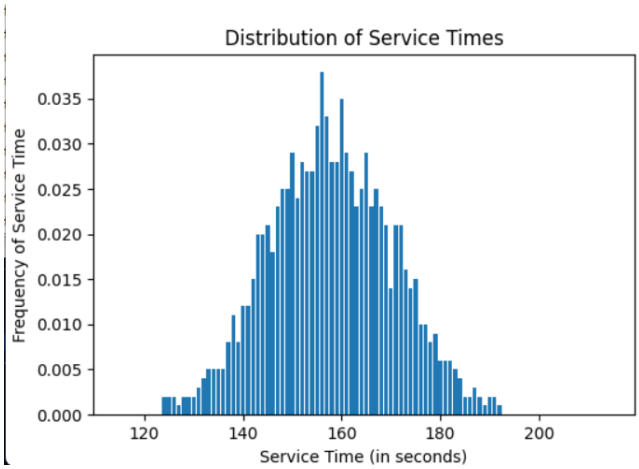

So, if *NumPy.random.poisson* works as intended, we can expect about 156 seconds to be the most common call length, happening about 3.5% of the time. Our plot is expanded upon in

the ***Plotting the Data*** section of this document, but here are some examples of the data we got from our simulation.

In an 8-hour simulation, the resulting plot of callLengths seems quite random, as there is a relatively low number of customers overall:




However, if we do a larger simulation, such as 100 hours, you can see the resulting bar chart is very similar to the actual Poisson distribution we modeled in Excel:




While the numbers are random, and the charts will vary with each simulation, the more hours that are simulated, the closer the line of best fit will be to the actual poisson distribution.

CallLengths is the only value that we visually plot in our project. However, callsPerDay and Customer.waitingTime are also generated using NumPy.random.poisson().

***Data Conversion***

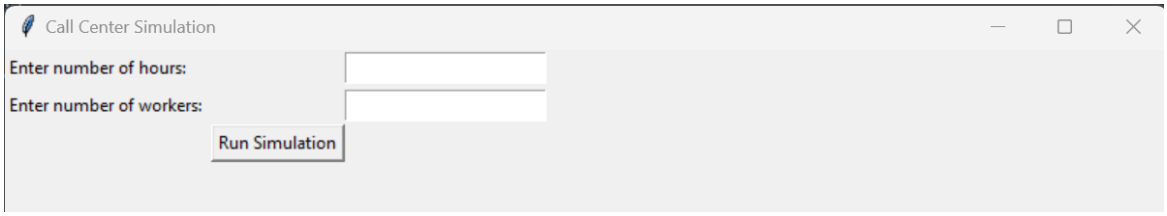
The dataset did not contain data in a format that was immediately usable by the program. *Talk Duration* and *Waiting Time* were formatted as time (i.e. 2 minutes is formatted as "2:00"), so the data was not as easy to manipulate for the purposes of this program. As such, we created a function to transform a time value into an integer, representing the number of seconds. Seconds is the unit of time that is used for most parts of this program, as you will see going forward.

```
## take in time string formatted as 00:00:00 and convert it to total seconds
def toSeconds(time):
    time = str(time)
    numbers = time.split(":")
    hours = int(numbers[0])
    minutes = int(numbers[1])
    seconds = int(numbers[2])
    return hours*3600 + minutes*60 + seconds
```

***Running the Simulation***

The goal of the simulation was to simulate a call center's performance over a certain period of time. We wanted to allow the user to be able to control the amount of time to be simulated, and the performance of the call center. We used our dataset to represent a standard 8-hour workday. However, we wanted the user to be able to simulate more time or less time, depending on the needs of the call center.

To run the program, the user must input a number of hours, **X**, to be simulated:




The program will then generate a random poisson number, based on the average calls per day from the dataset. This number will then be divided by 8 (breaking down the 8-hour day into 1-hour increments), and multiplied by the number of hours specified. Finally, it is rounded up to the nearest whole number, as any fraction left over should be treated as a full call.

```
def callsPerDay(hours):
    return int(np.ceil((np.random.poisson(avgCalls) / 8 ) * hours))
```

This leaves the program with a number representing how many calls could be expected in an **X**-hour period. We will call this number **Y**. The program then generates the time that each call takes place. To do this, it generates **Y** random numbers in the range 0 to 3600**X**, as there are 3600 seconds in an hour. This means calls can occur any second from the start of the program, to its conclusion. These values are stored in a list called *callTimes*. The *callTimes* list is sorted so that the times are chronological. Then, *callTimes* is used to generate another list called *callLengths*. *callLengths* is a list of random values from a Poisson distribution based on the average call length.

```
for i in callTimes:
    length = np.random.poisson(avgLength)
    callLengths.append(length)
```

Finally, a third list is created, *customerList*, which creates a list of *Customer* objects, which can hold a *callTime* and a *callLength*.

When the simulation is run, the *customerList* and the number of hours, **X,** are passed to a function called *simulate()*. *simulate()* takes the *customerList*. *simulate()* iterates **X** * 3600 times, and checks if the first customer in *customerList*'s *entryTime* matches the current iteration, or tick (each tick represents one second). If it does, it is added to a queue, to be served by a Call Center employee.

Each tick also calls the *tick()* function. The *tick()* function takes the queue, and checks for any updates. For example, if there are any unoccupied servers, and there is at least one person in the queue, the *tick()* function will remove the first person from the queue, and put them with the server. The *tick()* function also checks if each server is currently serving a customer. If the server's *endTime is* less than or equal to the current tick, the server is no longer considered to be occupied, and is ready to receive another customer. If the server's *endTime* is greater than the current tick, no change will occur, as the customer still needs to wait longer.

***Balking and Reneging***

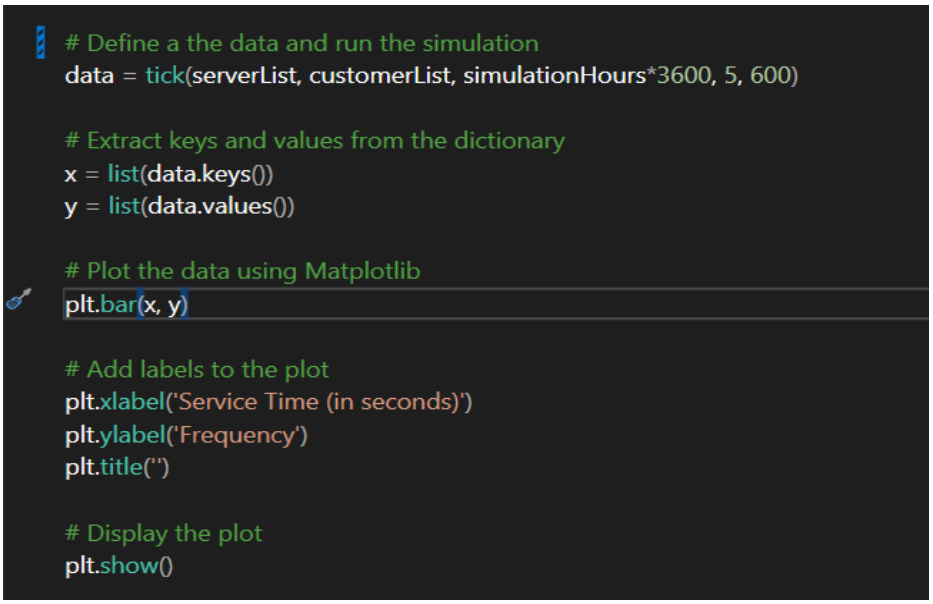
The simulation takes abandoned calls into account through balking and reneging. Balking in this simulation is a customer attempting to join the queue, but seeing the line is too long and deciding to not join. Reneging in this simulation is when a caller has been in the queue for too long and did not want to wait any longer to be served so they abandon the call. Through these cases, we were able to successfully implement elements of balking and reneging.

***Plotting The Data***

Matplotlib is a popular data visualization library in Python that can be used to create a wide range of graphs and plots. One area where matplotlib can be particularly useful is in call centers, where data is often collected and analyzed to improve customer service and efficiency.

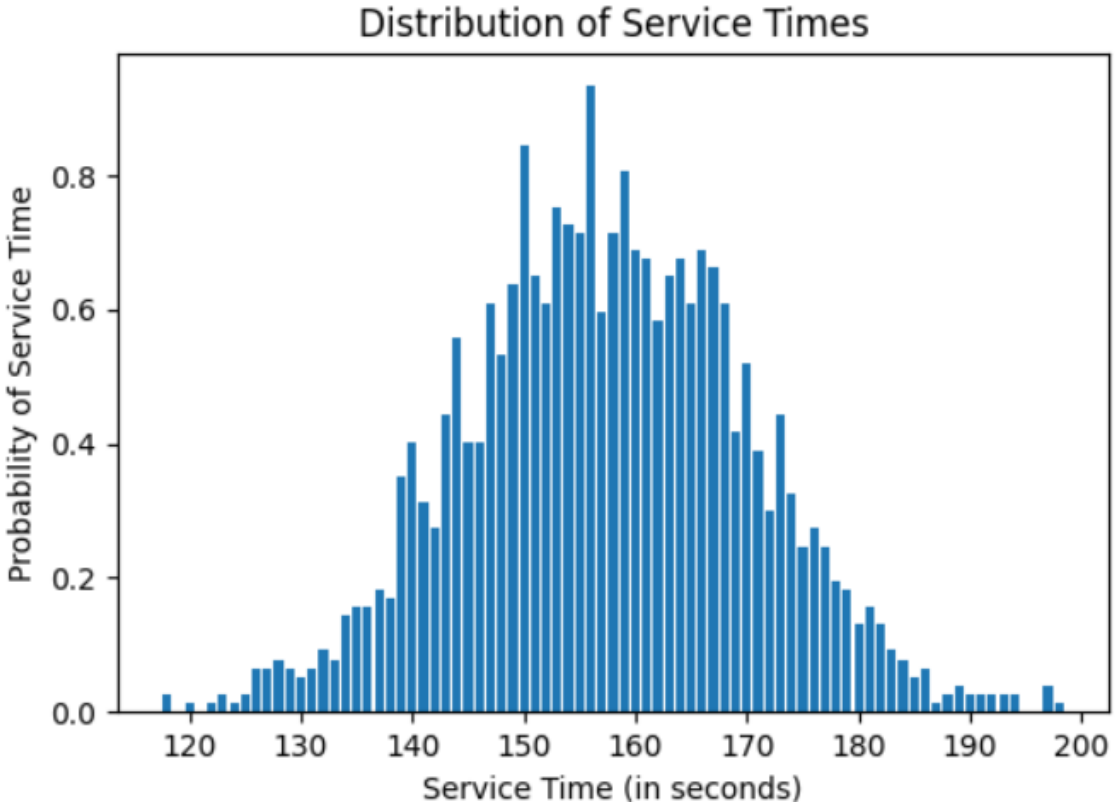
We utilized matplotlib for a call center, including examples of different types of graphs and how they can be used to gain insights into call center performance.

*Here is an example of how we utilized matplotlib to plot customer waiting time frequency:*



```
# Define a the data and run the simulation
data = tick(serverList, customerList, simulationHours*3600, 5, 600)

# Extract keys and values from the dictionary
x = list(data.keys())
y = list(data.values())

# Plot the data using Matplotlib
plt.bar(x, y)

# Add labels to the plot
plt.xlabel('Service Time (in seconds)')
plt.ylabel('Frequency')
plt.title('')

# Display the plot
plt.show()
```

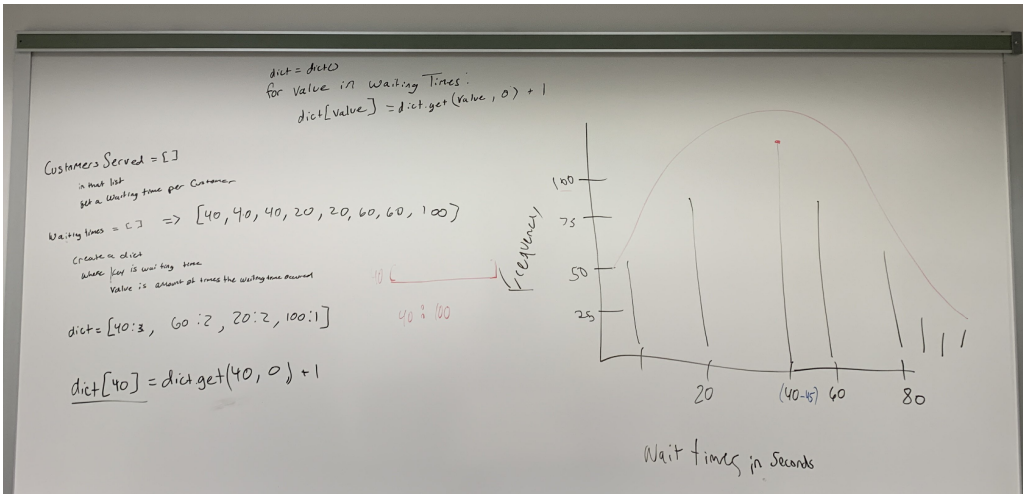
*The resulting graph after 100 hours and 1 server:*




***Gathering the Information to Send to Matplotlib***

To create data to model the service time, we had to record all of the customers who were served in a day and collect the time required to fulfill their needs. The pseudo-code below demonstrates the thought process behind gathering the data into a dictionary where the key was the service time and the value was the frequency in which that time occurred amongst all of the customers.

*Pseudo Code to gather the information to plot a Poisson Distribution:*




The code shown below further demonstrates how powerful Python is as a coding language. To count the frequency of a number in a list could have been several lines of code in many other languages, but in Python, it was a mere 4 lines of code. The way it functions is as follows, the code will create a list of all of the service times from a list of customers who were successfully served. The code will then create a dictionary and for each time in the service time list, it will add that service time to the dictionary or increment the number of times that service time has appeared in the dictionary. Utilizing this code block, we were able to gather the information needed to then calculate the probability of that service time occurring in that simulation. The next lines of code demonstrate adjusting the frequency of the service time to be the probability of that time occurring.

*Actual code used to gather the data for plotting:*

```
serviceTimeCounts = dict()
for customer in servedCustomers:
    serviceTime = customer.serviceTime
    serviceTimeCounts[serviceTime] = serviceTimeCounts.get(serviceTime, 0) + 1


for key in serviceTimeCounts.keys():
    serviceTimeCounts[key] = round(serviceTimeCounts.get(key) / len(servedCustomers), 3)
```

***GUI***

To start with implementing our GUI, we utilized Tkinter from Python to create everything we need to showcase our data. The first step was creating the window to fit our first appearance of what the user will see.

```
#set window size
WINDOW_LENGTH = 800
WINDOW_WIDTH = 300
window.geometry(str(WINDOW_LENGTH) + "x" + str(WINDOW_WIDTH))
```

Next, we had to create the text boxes and the button to input data and run the simulation. There are also labels on the side to show what should be inputted into the text boxes.

```
# Label for text boxes
lbl1 = Label(window, text = "Enter number of hours: ")
lbl1.grid(row = 0, column= 0, sticky=W, pady=2)

lbl2 = Label(window, text = "Enter number of workers: ")
lbl2.grid(row= 1, column=0, sticky=W, pady=2)
# ----------------------


# Text Box for Hours
entryTotalHours = Entry()
entryTotalHours.config(font=('Arial',12))
entryTotalHours.insert(0, "")
entryTotalHours.config(width=15)

# Text boc for Servers
entryTotalWorkers = Entry()
entryTotalWorkers.config(font=('Arial',12))
entryTotalWorkers.insert(0, "")
entryTotalWorkers.config(width=15)

# Text box locations
entryTotalHours.grid(row = 0, column=2, sticky=W, pady= 2)
entryTotalWorkers.grid(row = 1, column= 2, sticky=W, pady= 2)
```

Next, we created the submit button which stores the numbers that were given in the text boxes and also opens up a new window for the graph and table to be shown. We also linked the data that we utilized from main to display to the GUI system.

```
def submit():
    currentData = dict
    totalHours = entryTotalHours.get()
    totalWorkers = entryTotalWorkers.get()

    currentData = main(totalHours, totalWorkers)

    # create a new window
    graphWindow = Toplevel(window)
    graphWindow.title("Customer Count Plot")

    # create a figure object
    fig = plt.figure(figsize=(6, 4), dpi=100)

    # create a subplot
    ax = fig.add_subplot(111)

    # plot the data using the same code as in the main function
    x = list(currentData.keys())
    y = list(currentData.values())
    ax.bar(x, y)

    # Add labels to the plot
    ax.set_xlabel('Time (in seconds)')
    ax.set_ylabel('Number of Customers')
    ax.set_title('Frequency of Customer Count')

    # create a canvas to display the plot
    canvas = FigureCanvasTkAgg(fig, master=graphWindow)
    canvas.draw()
    canvas.get_tk_widget().pack()

    # display the window
    graphWindow.mainloop()
```

Lastly, the GUI will look like this when everything is set up, and all the data needed is entered.

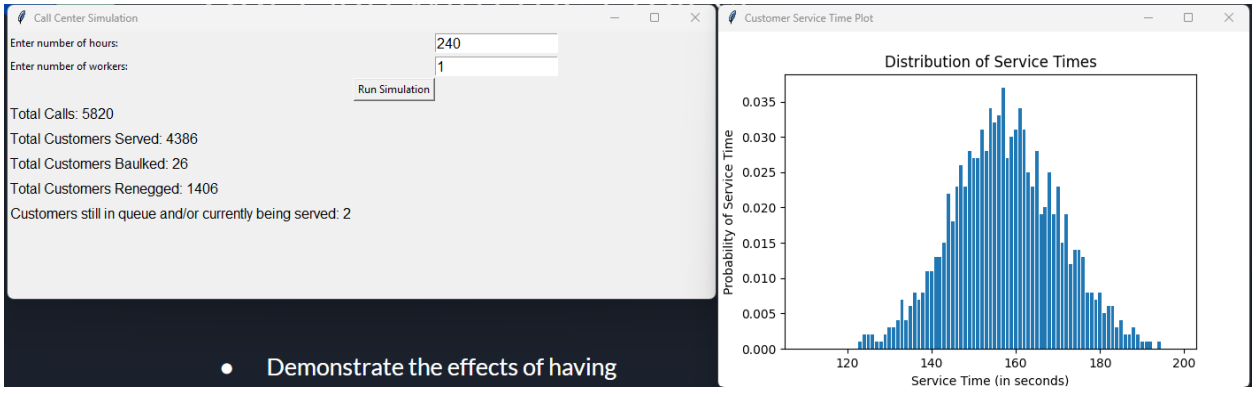
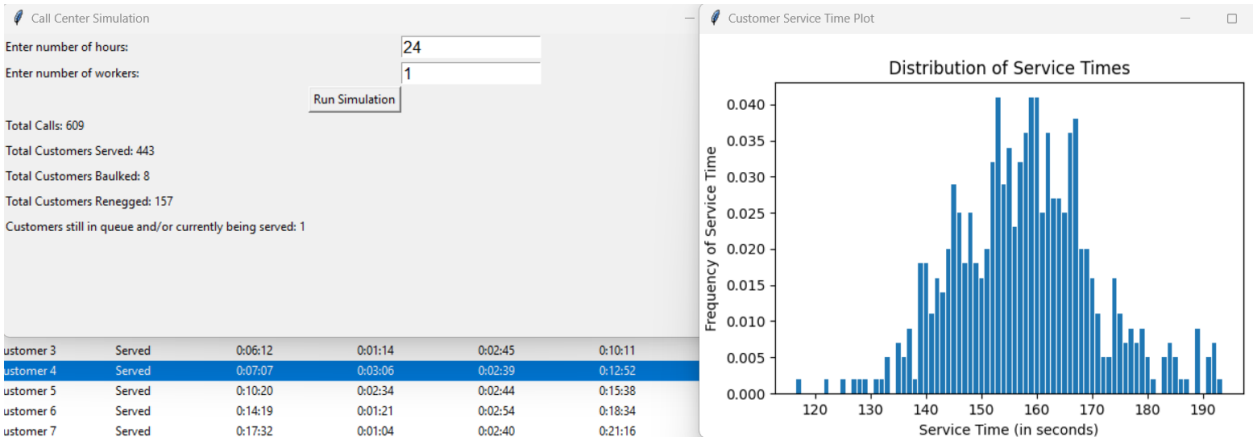


Table of Customers

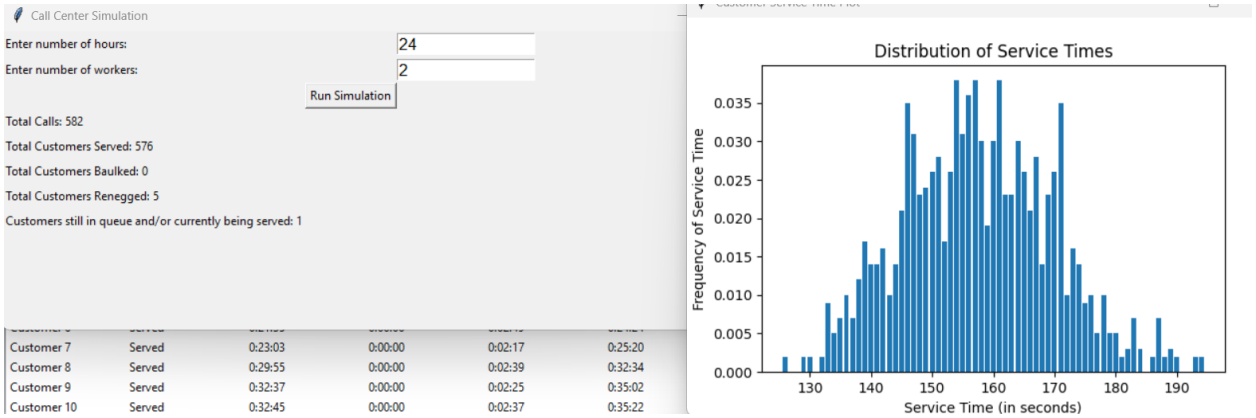
| Customer # | Status? | Entry Time | Waiting Time | Service Time | Exit Time |
|---|---|---|---|---|---|
| Customer 50 | Served | 1:42:22 | 0:01:46 | 0:02:38 | 1:46:46 |
| Customer 51 | Served | 1:53:12 | 0:00:00 | 0:02:37 | 1:55:49 |
| Customer 52 | Served | 1:53:45 | 0:02:06 | 0:02:39 | 1:58:30 |
| Customer 53 | Served | 1:56:13 | 0:02:19 | 0:02:42 | 2:01:14 |
| Customer 54 | Balked | 2:02:25 | 0:00:01 | 0:00:00 | 0:00:00 |
| Customer 55 | Reneged | 1:59:10 | 0:03:40 | 0:00:00 | 0:03:40 |
| Customer 56 | Served | 1:58:15 | 0:03:01 | 0:02:42 | 2:03:58 |
| Customer 57 | Reneged | 2:00:39 | 0:04:06 | 0:00:00 | 0:04:06 |
| Customer 58 | Reneged | 2:00:28 | 0:04:28 | 0:00:00 | 0:04:28 |
| Customer 59 | Reneged | 2:02:16 | 0:04:21 | 0:00:00 | 0:04:21 |

## Results

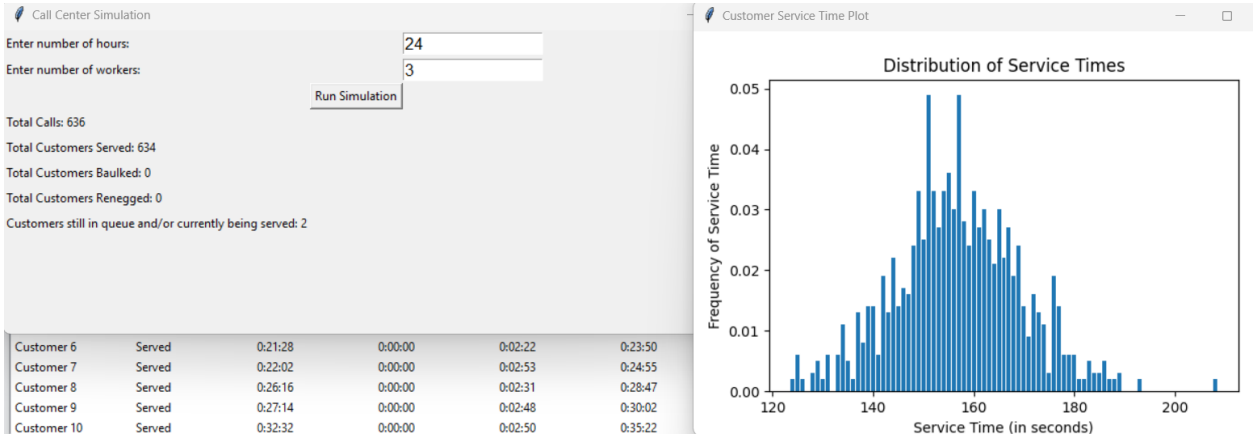
After testing the simulation to run with 24 hours (3 8-hour work days) and only one server, the program will display the distribution of the service times of the callers who did get served and the number of callers who did not see their call through to the end. In additional tests, like the ones shown. The results show that with a single server, there will be a significant number of callers who renege from the queue and a handful will balk.




When two servers are being used the number of balks and reneges drastically decreases. However, there are still a few balks and reneges that can occur.




When 3 or more servers are used, the call center's efficiency becomes almost perfect - in multiple simulations, we did not experience a single balk or renege, as the servers had the capacity to handle every single caller.




As call times are randomly generated, it is still possible for too many callers to join at the same time, during a 3-worker simulation, resulting in a balk or renege. However, the probability of this seems near 0 from our simulations with this number of workers.

## Future Implementation

For future implementations of this program, we can add more adjustability to the program's input data. For example, most of the background calculations were based on the data set we got from Kaggle. We can expand the GUI to take in any additional information a company

or a call center can provide. This will change how the program runs and will allow multiple different industries to use the program to test the effectiveness of their centers.

Additionally, we could make the simulation more accurate, by pulling even more data from the csv file. For example, one of the columns in the csv file is “Answer Speed” - the time it takes a server to answer a call, when that call is in queue. We chose not to include this in our simulation because the values are overall very low, so we just had the servers answer instantaneously. However, adding this value, as well as some of the values from the other columns, could help improve the accuracy of the simulation.

Another possible addition could be a way for the user to manually step through the program. At the moment, if you simulate 8-hours, the simulation would spit out the data for all 8-hours. However, adding a way for the user to step 15 minutes at a time through an 8-hour simulation may be valuable.

## Conclusion

From our project, we can conclude that a call center must have the proper amount of staff to function properly. In the figures shown in the ***Results*** section, you can see that a center loses mass amounts of revenue depending on the number of servers the center has. In that instance of having only one server, the center experienced a high number of balking and reneging callers. Two employees is almost perfect - and may be the most cost-efficient option for this call-center, however there will still be a few reneging customers. If the call center truly wants to serve every single caller, our data shows that 3 or more servers would be best. These callers that left the queue early or did not want to join it in scenarios with only one server could have been big spenders for the company. It is entirely possible that the caller was someone who was just inquiring about the business and did not intend to buy anything, but these people can leave bad

reviews about the customer service of the company. With bad reviews, the company's rating can falter and cause them to lose business. With that in mind, a company must be equipped to handle the inflow of their customers in their call center to avoid any issues that could arise from improper staffing. Ultimately, we discovered that with the dataset we used, only one employee was far too few - there would be a significant number of customers lost.

**References**

Branka. (2023, January 9). *Call center statistics to know in 2023*. TrueList. Retrieved April 27, 2023, from https://truelist.co/blog/call-center-statistics/

Estrellado, V. (2023, March 14). *Call center statistics: What to expect on the industry in 2023*. Outsource Accelerator. Retrieved April 27, 2023, from https://www.outsourceaccelerator.com/articles/call-center-statistics/

Jahromi, M. A. (2021). Call Center Data. Retrieved March 8, 2023, from https://www.kaggle.com/datasets/satvicoder/call-center-data?select=Call+Center+Data.csv.

LiveAgent. (2023, March 15). *Call center ^benchmarks^*. LiveAgent. Retrieved April 27, 2023, from https://www.liveagent.com/research/call-center-benchmarks/#:~:text=To%20break%20this%20statistic%20down,the%20industry%20you're%20in.