a)

Low Utilization:
Results from running the p parameter 10 times each for FIFO and SJN:

FIFO

Index	Elapsed (wall clock) time (m:ss.ms)
1	3:42.36
2	2:51.06
3	2:35.44
4	2:34.43
5	2:36.22
6	2:35.98
7	2:34.13
8	2:34.94
9	2:58.67
10	2:33.90

(mmx60) + ss for all 10
FIFO Average = 165.713 seconds

SJN

Index	Elapsed (wall clock) time (m:ss.ms)
1	2:57.40
2	2:34.77
3	2:34.23
4	2:45.44

5	2:40.57
6	2:55.86
7	2:36.05
8	2:34.27
9	2:34.89
10	2:36.94

(mmx60) + ss for all 10
SJN Average = 161.042 seconds

High Utilization:
Results from running the p parameter 10 times each for FIFO and SJN:
FIFO

Index	Elapsed (wall clock) time (m:ss.ms)
1	1:43.59
2	0:41.94
3	0:46.31
4	0:42.35
5	0:41.87
6	0:41.29
7	0:40.68
8	0:44.37
9	0:40.48
10	0:40.98

(mmx60) + ss for all 10
FIFO Average = 48.386 seconds

SJN

Index	Elapsed (wall clock) time (m:ss.ms)
1	0:46.45
2	0:40.24
3	0:41.20
4	0:41.40
5	0:41.87
6	0:40.37
7	0:40.32
8	0:41.58
9	0:40.54
10	0:42.15

(mmx60) + ss for all 10
SJN Average = 41.512

Explanation:

In low utilization, the average runtimes for both FIFO and SJN are nearly identical, which is as expected as we saw in lecture that scheduling doesn't really matter at lower utilizations, with only a marginal improvement for SJN (161.042 < 165.13). Re-ordering does little to improve utilization since the resources are under low loads.

However, as expected, In high utilization, SJN outperforms FIFO (41.412 < 48.386) as it optimizes for shorter jobs, which leads to lower average runtimes. The time it takes to re-order significantly increases the overall improvement, so the slowdown due to becomes negligible. T

b)

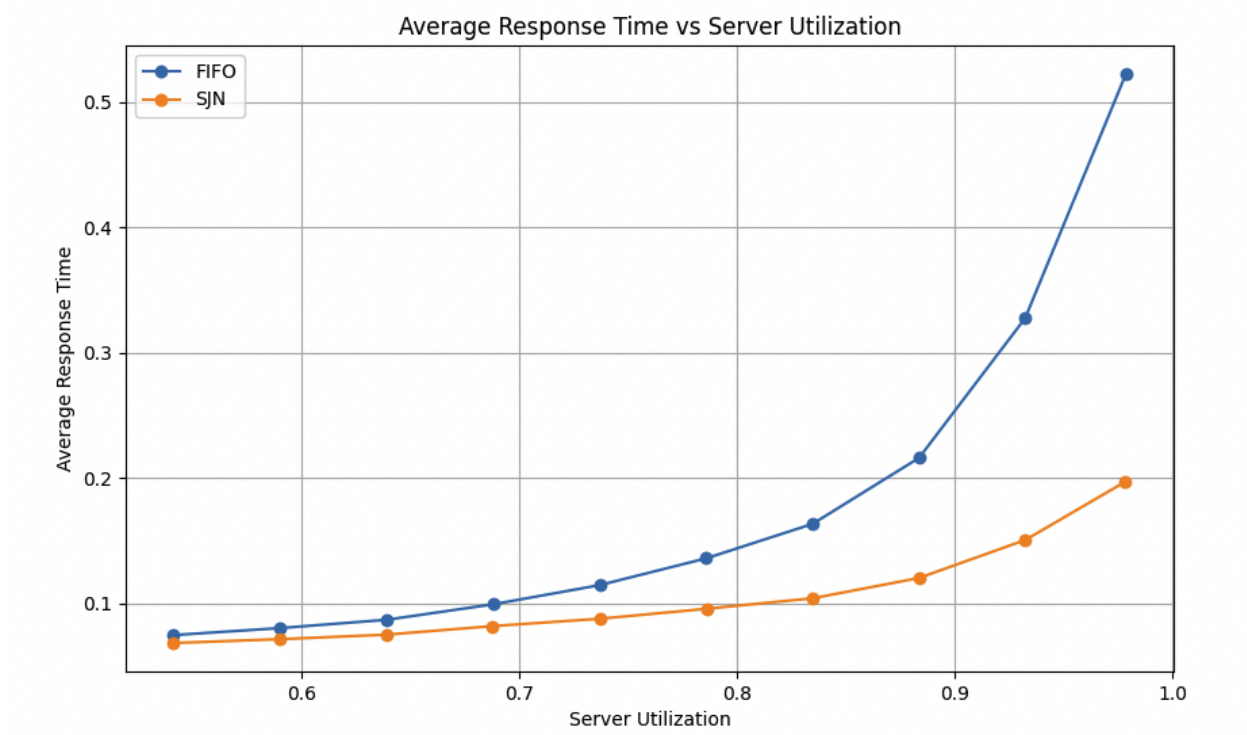
Here, I first wrote a simple bash script to automate the server's running multiple times. After I collated all the logs, I then ran the below python file, which computes the overall busy time and divides the time the server was active, which totals the time each worker was busy.

```
import os
import re
import numpy as np
import matplotlib.pyplot as plt

# Parameters
POLICIES = ["FIFO", "SJN"]
ARRIVAL_RATES = [22, 24, 26, 28, 30, 32, 34, 36, 38, 40]

WORKERS = 2

# Initialize data structures
data = {policy: {'utilization': [], 'response_time': []} for policy in POLICIES}

# Regular expressions to parse log files
request_pattern =
re.compile(r'R(\d+):([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+)')
worker_pattern = re.compile(r'T(\d+)
R(\d+):([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+)')

for policy in POLICIES:
   for arr_rate in ARRIVAL_RATES:
       server_log_path = f'Eval_B_logs/{policy}/server_{arr_rate}.log'
       worker_busy_times = {}
       total_response_time = 0.0
       num_requests = 0
       start_times = []
       completion_times = []

       with open(server_log_path, 'r') as f:
           for line in f:
               # Match lines indicating request completion
               match = worker_pattern.match(line.strip())
               if match:
                   # Extract data
                   worker_id = int(match.group(1))
                   req_id = int(match.group(2))
                   req_timestamp = float(match.group(3))
                   req_length = float(match.group(4))
                   receipt_time = float(match.group(5))
                   start_time = float(match.group(6))
                   completion_time = float(match.group(7))
```

```
                # Calculate busy time for this request
                busy_time = completion_time - start_time

                # Update worker busy times
                if worker_id not in worker_busy_times:
                    worker_busy_times[worker_id] = 0.0
                worker_busy_times[worker_id] += busy_time

                # Calculate response time
                response_time = completion_time - req_timestamp
                total_response_time += response_time
                num_requests += 1

                start_times.append(start_time)
                completion_times.append(completion_time)

        # Calculate total busy time
        total_busy_time = sum(worker_busy_times.values())

        # Calculate total runtime
        if completion_times:
            total_runtime = max(completion_times) - min(start_times)
            total_capacity = total_runtime * WORKERS
            server_utilization = total_busy_time / total_capacity
        else:
            server_utilization = 0.0

        # Calculate average response time
        if num_requests > 0:
            avg_response_time = total_response_time / num_requests
        else:
            avg_response_time = 0.0

        data[policy]['utilization'].append(server_utilization)
        data[policy]['response_time'].append(avg_response_time)
        print(f"Policy: {policy}, Arrival Rate: {arr_rate}, Utilization:
{server_utilization:.4f}, Avg Response Time: {avg_response_time:.4f}")

# Plotting
plt.figure(figsize=(10, 6))
```

```
for policy in POLICIES:
   plt.plot(data[policy]['utilization'], data[policy]['response_time'], marker='o',
label=policy)

plt.xlabel('Server Utilization')
plt.ylabel('Average Response Time')
plt.title('Average Response Time vs Server Utilization')
plt.legend()
plt.grid(True)
plt.savefig('response_time_vs_utilization.png')
plt.show()
```

Output:



So we can clearly see here that FIFO and SJN perform relatively similarly for lower utilization rates, but as the server utilization increases, the average response rate for FIFO grows significantly faster (almost exponentially) compared to SJN. At max utilization, the average response time is about 0.3 seconds faster for SJN. FIFO seems to be 1.5x slower than SJN at max utilization (0.5-0.2/0.3)

C)

Overview of Steps Taken:

1. **Extract Response Times**: Parse the server log files from the 10th run for both FIFO and SJN policies to extract the response times of each request.
2. **Calculate Statistics**: Compute the average response time and the 99th percentile response time for each policy.
3. **Generate CDF Plots**: Plot the CDF of the response times for each policy so that both plots have the same x and y axis ranges for visual clarity.
4. **Add Vertical Lines**: On each plot, add vertical lines to mark the average response time and the 99th percentile response time.

Python Script:

```
import re

import matplotlib.pyplot as plt

import numpy as np



# Define the policies and the corresponding log files

policies = {

   'FIFO': 'EVAL_B_logs/FIFO/server_40.log',

   'SJN': 'EVAL_B_logs/SJN/server_40.log'

}



# Regular expression to parse the log lines

log_pattern = re.compile(

   r'T(\d+) R(\d+):([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+)'

)



# Dictionary to store response times for each policy
```

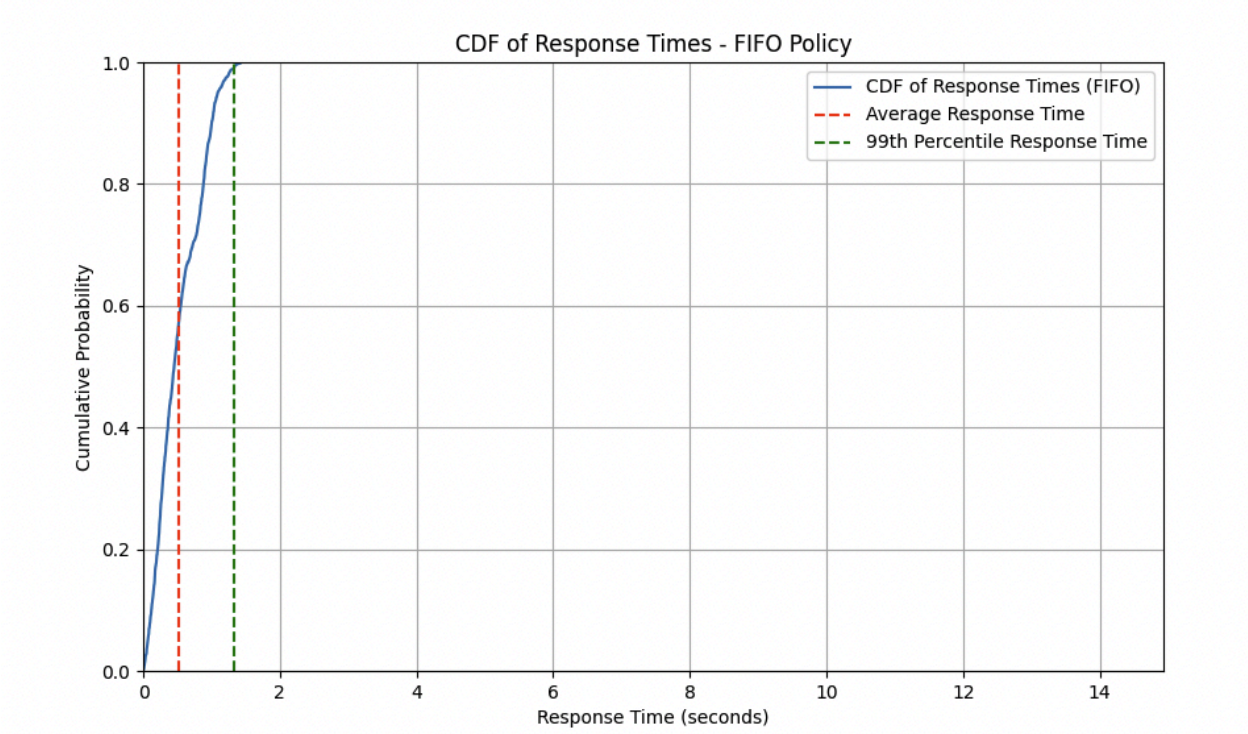
```
response_times = {}



for policy, log_file in policies.items():

   response_times[policy] = []



   with open(log_file, 'r') as f:

       for line in f:

           match = log_pattern.match(line.strip())

           if match:

               # Extract the timestamps

               req_timestamp = float(match.group(3))

               completion_time = float(match.group(7))



               # Calculate response time

               resp_time = completion_time - req_timestamp



               # Append to the list

               response_times[policy].append(resp_time)



   print(f"Policy: {policy}, Number of Requests: {len(response_times[policy])}")



# Dictionary to store statistics

stats = {}
```

```
for policy in policies.keys():
    times = response_times[policy]
    times.sort()  # Sort the response times

    # Calculate average response time
    avg_response_time = np.mean(times)

    # Calculate 99th percentile response time
    percentile_99 = np.percentile(times, 99)

    # Store in stats dictionary
    stats[policy] = {
        'average': avg_response_time,
        'percentile_99': percentile_99,
        'sorted_times': times  # Store sorted times for CDF plot
    }

    print(f"Policy: {policy}")
    print(f"  Average Response Time: {avg_response_time:.4f} seconds")
    print(f"  99th Percentile Response Time: {percentile_99:.4f} seconds")

# Function to plot the CDF
def plot_cdf(times, avg_time, percentile_99, policy_name, x_limits, y_limits):
    # Calculate the CDF values
```

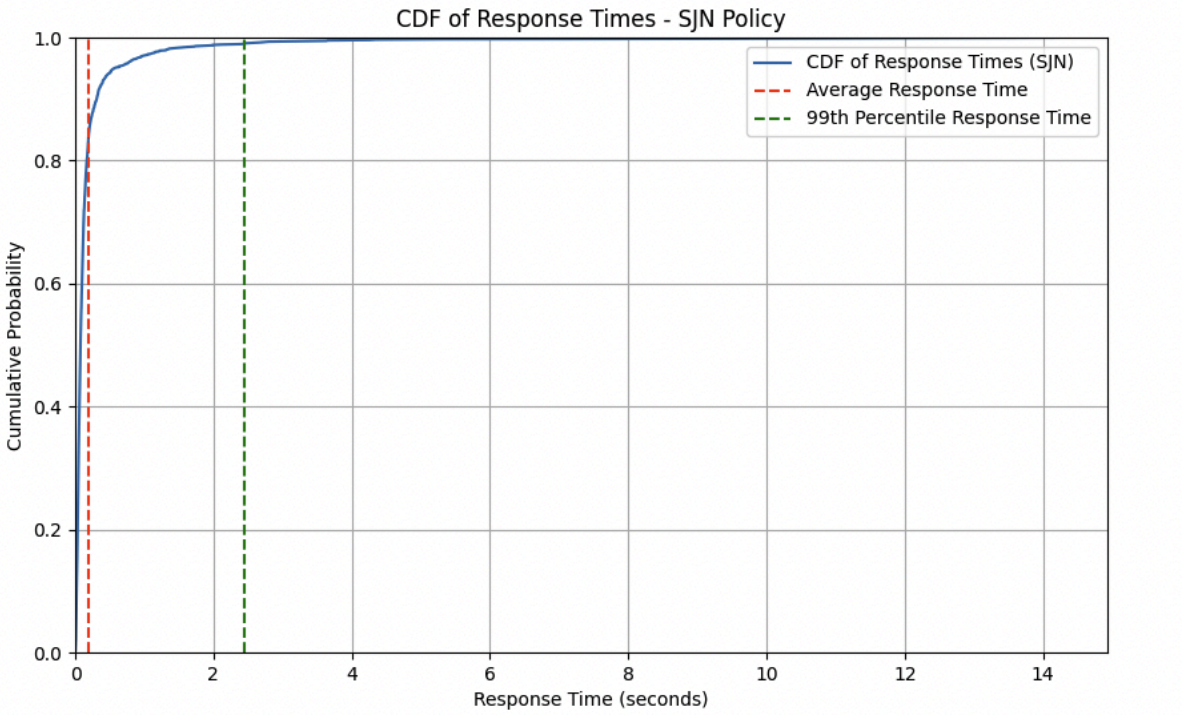
```
    sorted_times = np.sort(times)

    cdf = np.arange(1, len(sorted_times)+1) / len(sorted_times)



    # Plot the CDF

    plt.figure(figsize=(10, 6))

    plt.plot(sorted_times, cdf, label=f'CDF of Response Times ({policy_name})')



    # Add vertical lines for average and 99th percentile

    plt.axvline(x=avg_time, color='r', linestyle='--', label='Average Response Time')

    plt.axvline(x=percentile_99, color='g', linestyle='--', label='99th Percentile
Response Time')



    # Set the x and y limits to make plots comparable

    plt.xlim(x_limits)

    plt.ylim(y_limits)



    # Labels and Title

    plt.xlabel('Response Time (seconds)')

    plt.ylabel('Cumulative Probability')

    plt.title(f'CDF of Response Times - {policy_name} Policy')

    plt.legend()

    plt.grid(True)



    # Save the plot
```

```
    plt.savefig(f'cdf_response_times_{policy_name.lower()}.png')
    plt.show()


# Determine common x and y limits based on combined data for comparability
all_times = response_times['FIFO'] + response_times['SJN']
x_min = 0
x_max = max(all_times) * 1.05  # Slightly greater than max to give some space
y_min = 0
y_max = 1


x_limits = (x_min, x_max)
y_limits = (y_min, y_max)


# Plot for each policy
for policy in policies.keys():
    times = stats[policy]['sorted_times']
    avg_time = stats[policy]['average']
    percentile_99 = stats[policy]['percentile_99']


    plot_cdf(times, avg_time, percentile_99, policy, x_limits, y_limits)
```

Output:



Policy: FIFO, Number of Requests: 1500
Policy: FIFO
Average Response Time: 0.5223 seconds
99th Percentile Response Time: 1.3252 seconds



Policy: SJN, Number of Requests: 1500
Policy: SJN
Average Response Time: 0.1970 seconds
99th Percentile Response Time: 2.4490 seconds

D)

FIFO CDF Plot:

- The average response time is quite close to the 99th percentile response time.
- This indicates that most requests experience fairly similar response times, with fewer outliers or extreme values.
- The tail is shorter, meaning fewer requests experience significantly longer delays.
- This makes FIFO a more predictable policy in terms of uniformity across all requests. Even though FIFO may not optimize the fastest response times, it does ensure that the majority of requests experience relatively similar delays.

SJN CDF Plot:

- The 99th percentile response time is much higher than the average response time, indicating that while many requests are processed quickly, some outliers experience very long response times.
- This results in a longer tail for SJN, meaning that while the majority of requests are handled faster than FIFO, a small percentage experience delays.

- SJN prioritizes shorter requests, meaning that while most requests are handled very quickly (faster than in FIFO), the system is less predictable because some longer requests experience significant delays.

In regards to the definition of predictability in the textbook, the difference between the best-case and worst-case behavior is slight, indicating a more predictable system under FIFO.