



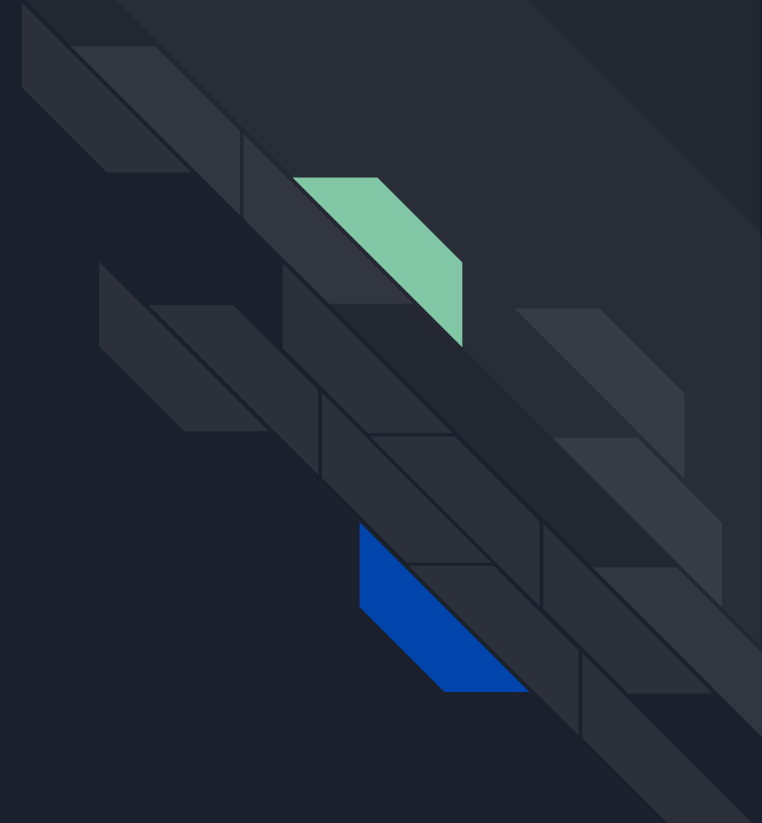
# Simulator in JS

Q 4.20

Presented by: Group A7

# TOC

1. Problem Understanding
2. System Boundary and Scope
3. UML Diagram
4. Solution
5. Program Flow
6. Alternate Models - Comparison
7. Alternate Solution
8. Future Enhancements





# 1. Problem Understanding

## 1.1 Problem Objective

Discrete Event Simulation (DES) is a technique used to model systems where events occur at specific points in time rather than continuously. In this case, we are simulating an airline ticket counter, where customers arrive, wait in line if necessary, and complete their transactions.

The **key challenge** is managing these events efficiently using an **event queue**, which processes events in time order. The goal is to analyze customer service performance by computing metrics like **average wait time**.

## 1.2 Problem Statement

We are simulating a ticket counter where:

- Customers **arrive** at random times.
- If an agent is available, the customer is **served immediately**.
- If all agents are busy, the customer **waits in a queue**.
- Each transaction **takes 3 minutes** to complete.
- Once a transaction is complete, the agent either serves the next waiting customer or becomes free.
- The simulation continues until all events have been processed, and then we calculate the **average customer wait time**.



### 1.3 Goal

- Efficiently **process all customer arrivals and transactions**.
- Familiarize with **event-driven approach**.
- **Minimize customer wait time** by optimizing agent allocation.
- **Collect and analyze data**, such as average waiting time, number of busy agents, and queue length.

### 1.4 Aspects Covered

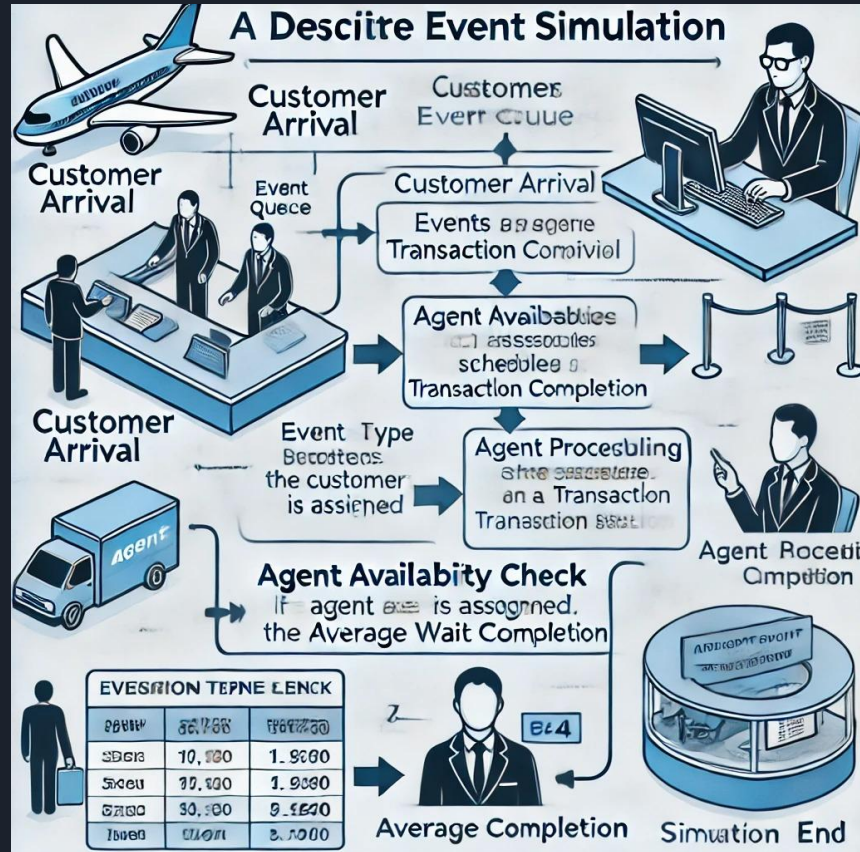
- **Event Handling:** Customer arrival and transaction completion.
- **Queue Management:** Managing waiting customers efficiently.
- **Resource Allocation:** Tracking agent availability.
- **Performance Metrics:** Analyzing system efficiency.

### 1.5 Approach

- Use an **Event Queue (Priority Queue)** to manage events in chronological order.
- Implement a **waiting line structure** for customers who cannot be served immediately.
- Maintain a **counter for busy agents** to track availability.
- Process events one by one, updating the system state accordingly.
- At the end, compute **performance metrics**, such as average wait time.

### 1.6 Constraints

- Each transaction takes exactly **3 minutes**.
- The number of agents is **dynamic**.
- Customers **arrive at predetermined times** (input data).
- The event queue **must always be ordered by time** (priority queue).
- The simulation ends **only when all events have been processed**.





## 2. System Boundary and Scope:


The **system boundary** defines what is **inside the system** (controlled by the simulation) and what is **outside the system** (external interactions and dependencies).

### 1. Internal Entities (Objects Within the System)

These are objects and components that are part of the **system logic** and operate inside the boundary.

### 2. External Entities (Objects Interacting with the System)

These are **external components** that interact with the system **through user input and output**.





## 2.1 Internal Entities Contributing to Waiting Time Calculation:

<u>Entity</u>	<u>Role in Waiting Time Calculation</u>
<b>Event</b> (Internal)	Represents an individual event (customer-arrival or transaction-complete). Each event <b>has a time</b> , which is used in waiting time calculations.
<b>EventQueue</b> (Internal)	Orders and processes events in time sequence. Ensures <b>arrival and transaction complete events are processed correctly</b> .
<b>waitingLine</b> (Internal)	Holds customers <b>who are waiting for an agent</b> . The <b>first customer in the queue is served next</b> , impacting waiting time.
<b>busyAgents</b> (Internal)	Tracks the number of currently busy agents. If all agents are busy, <b>waiting time increases</b> for new arrivals.
<b>waitTimes</b> (Internal)	Stores the waiting time for each customer. Used to <b>calculate the average and max waiting times</b> .



## 2.2 External Entities Contributing to Waiting Time Calculation:

<u>Entity</u>	<u>Role in Waiting Time Calculation</u>
<b>User Input (#arrivals)</b> (External)	Determines the <b>arrival times</b> of customers, which directly affects waiting time.
<b>Simulation Log (#log)</b> (External)	Stores time-based logs of when customers arrive, wait, and get served, indirectly verifying waiting time calculations.
<b>Chart.js (#waitTimeChart)</b> (External)	Visualizes customer wait times but does not affect calculations.



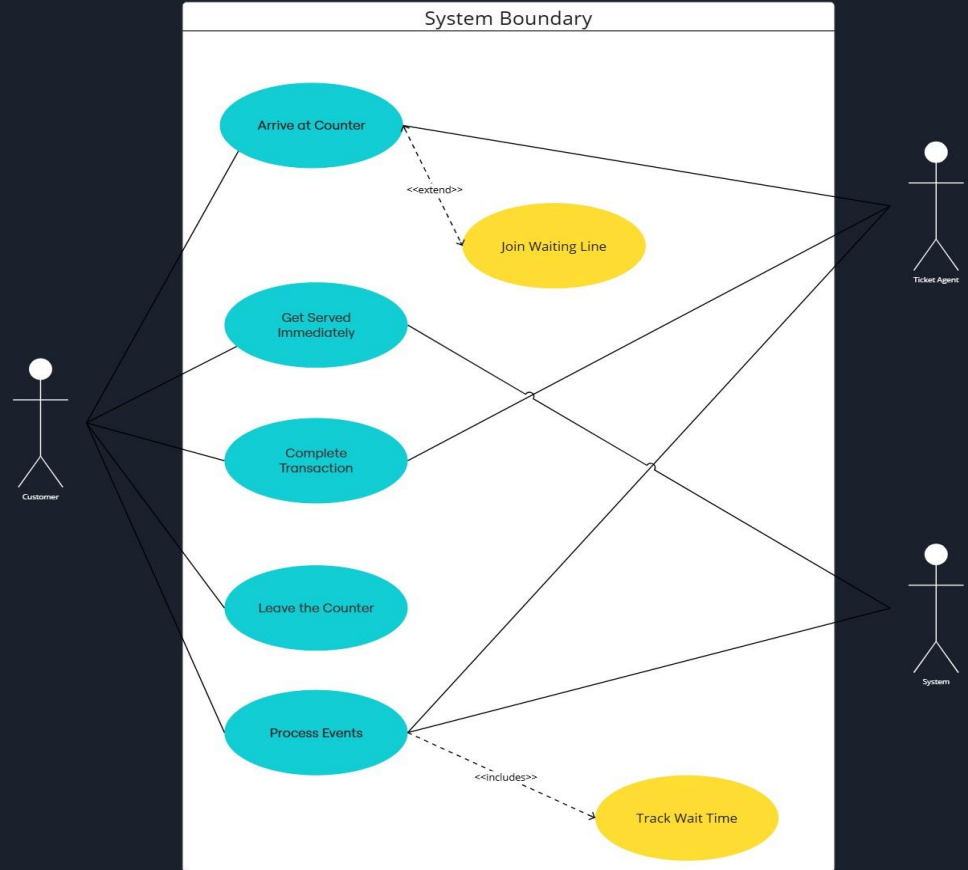


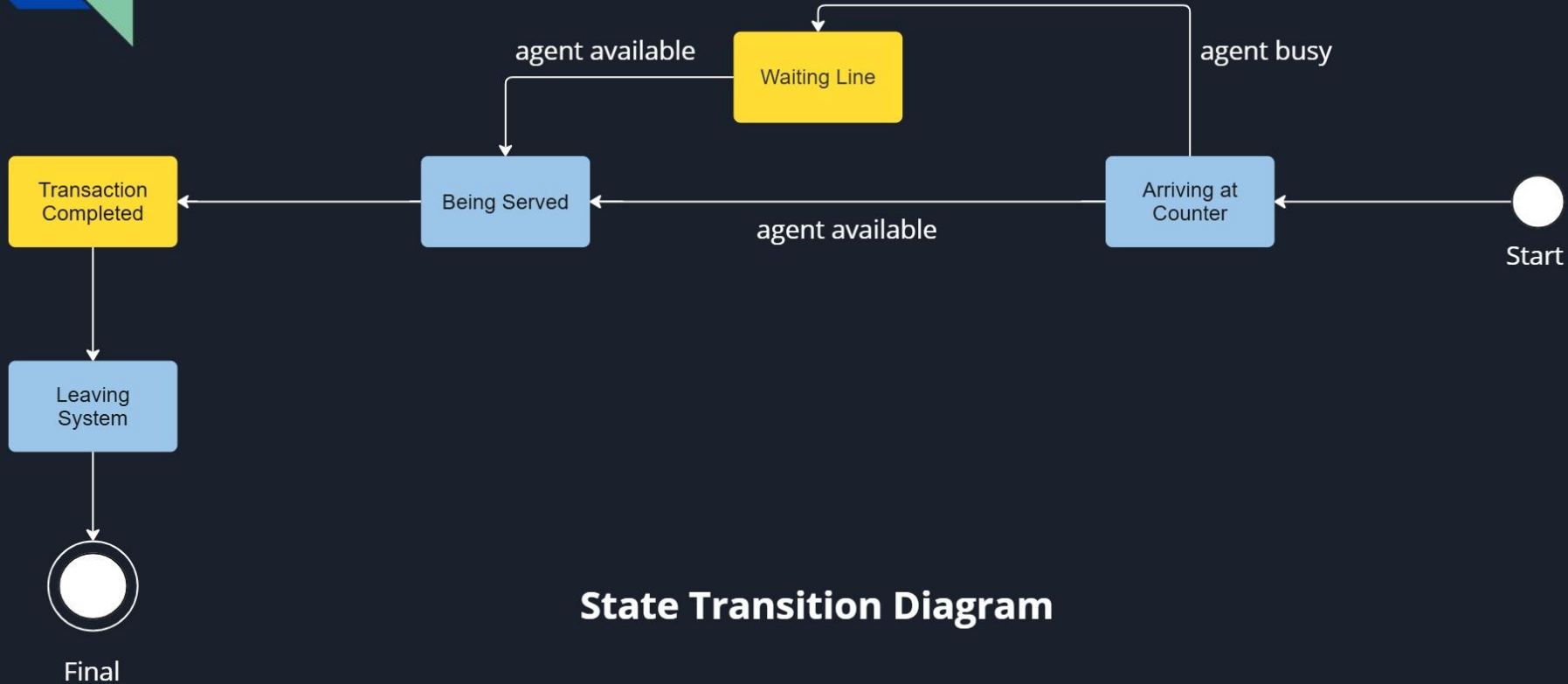
## 2.3 Relationship Between Entities:

1. **Customer Arrival** → `EventQueue` processes `customer-arrival` event.
  - If an agent is free (`busyAgents < numAgents`), customer is served immediately (**Waiting time = 0**).
  - If all agents are busy, customer joins `waitingLine`.
2. **Transaction Completion** → `EventQueue` processes `transaction-complete` event.
  - If `waitingLine` has customers, the next one is served (**Waiting time = Current time - Arrival time**).
  - If `waitingLine` is empty, an agent becomes free (`busyAgents--`).
3. **Final Calculation**
  - **Average Wait Time** = Total `waitTimes` ÷ Number of customers.
  - **Max Wait Time** = Highest value in `waitTimes`.

### 3. UML DIAGRAMS

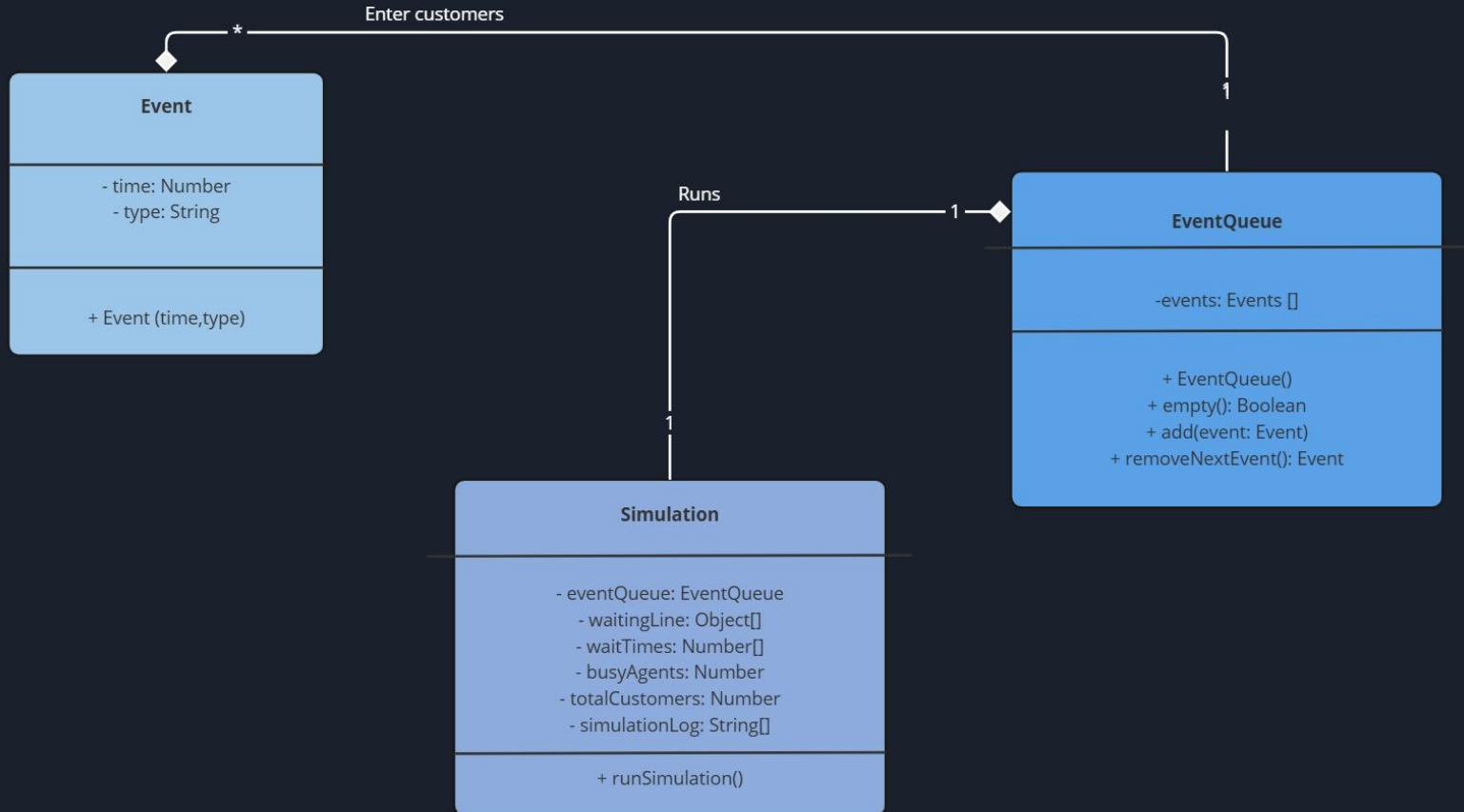
## Use Case Diagram





**State Transition Diagram**

# Class Diagram





## 4. SOLUTION

### 4.1 Mathematical Analysis

#### FORMULAS

- Average Wait Time =  $\frac{\Sigma \text{ Wait Time}}{\text{Number of Customers}}$
- Service Rate = 3 Minutes (Hardcoded)
- Queue Length = Arrivals – (Agents \* Time)

### 4.2 Data Structures

STRUCTURE	REASON	IMPLEMENTATION
Event Queue	Ensures chronological processing of events	Priority Queue (Array + Insertion Sort)
Waiting Line	Models real-world FIFO queuing behavior	Array (FIFO)
Wait Time	Efficiently stores and calculates performance metrics	Array
Simulation Log	Simple and effective for recording and displaying event details	Array of Strings
Event (Object)	structured representation of events.	Plain JS Class Constructor



## 4.3 Pseudocode

```
EVENT {  
  time: when it happens  
  type: type of event  
}  
  
QUEUE {  
  events: ordered list by time  
  operations: add, remove, check empty  
}  
  
WHILE (events exist) {  
  IF (customer arrives)  
    IF (agent available)  
      schedule completion  
    ELSE  
      add to waiting line  
  
  IF (transaction completes)  
    IF (customers waiting)  
      serve next customer  
    ELSE  
      free up agent  
}  
  
CALCULATE_STATISTICS:  
  compute average wait time  
  display results
```

## 4.4 Implementation Strategy

- Phase 1: Core Classes

```
class Event {  
  constructor(time, type) {  
    this.time = time;  
    this.type = type;  
  }  
}  
  
class EventQueue {  
  // Priority queue implementation  
}
```

- Phase 2: Simulation Logic

- runSimulation()

```
function runSimulation() {  
  const eventQueue = new EventQueue();  
  const waitingLine = [];  
  let busyAgents = 0;  
  
  while (!eventQueue.empty()) {  
    processNextEvent();  
  }  
}
```

- processNextEvent()

```
if (event.type === 'customer-arrival') {  
  handleCustomerArrival();  
} else if (event.type ===  
'transaction-complete') {  
  handleTransactionComplete();  
}
```

- updateStatistics()

```
const averageWait = waitTimes.reduce((a, b) => a  
+ b, 0) / waitTimes.length;  
const maxWait = Math.max(...waitTimes);
```



## 4.5 Enhanced Features

### Visualization

- Integrated Chart.js for real-time wait time graphs
- Real-time event logging for better insights

### User Interface

- Input validation for robust user control
- Dynamic statistics updates reflecting current queue state
- Clean, responsive layout (CSS styling)

## 4.6 Testing Approach

### Scenarios

1. Single agent, multiple customers (tests basic queue functionality)
2. Multiple agents, sparse arrivals (ensures concurrency handling)
3. Edge Cases:
  - No customers (system remains idle)
  - All customers arrive at once (stress test)



## 5. Program Flow: Discrete Event Simulation for Airline Ticket Counter

### 5.1 Overview of the Simulation Process

The program simulates an airline ticket counter operation using a discrete event simulation model. The main components include:

- **Event Queue:** A priority queue where events are processed in order of occurrence.
- **Customer Handling:** Customers arrive, either get served immediately or enter a waiting line.
- **Agent Availability:** If an agent is available, the customer is served; otherwise, they wait.
- **Transaction Completion:** When a transaction is completed, an agent becomes free and serves the next customer.
- **Simulation Termination:** Ends when no events remain in the queue.





## 5.2 Step-by-Step Program Flow

### Step 1: Initialization

- Read input: Number of ticket agents and customer arrival times.
- Initialize EventQueue with customer arrival events.
- Initialize Agent Count and Waiting Line.

### Step 2: Processing Events from the Queue

#### Event Type 1: Customer Arrival

- If a ticket agent is free:
  - Assign agent.
  - Schedule Transaction Complete event (current time + 3 minutes).
- Else:
  - Customer joins the Waiting Line.



### Event Type 2: Transaction Completion

- If customers are in line:
  - Remove first customer from the Waiting Line.
  - Record wait time.
  - Schedule a Transaction Complete event.
- Else:
  - Agent becomes free.

### Step 3: End of Simulation

- Continue processing events until EventQueue is empty.
- Compute average wait time for customers.
- Output results.

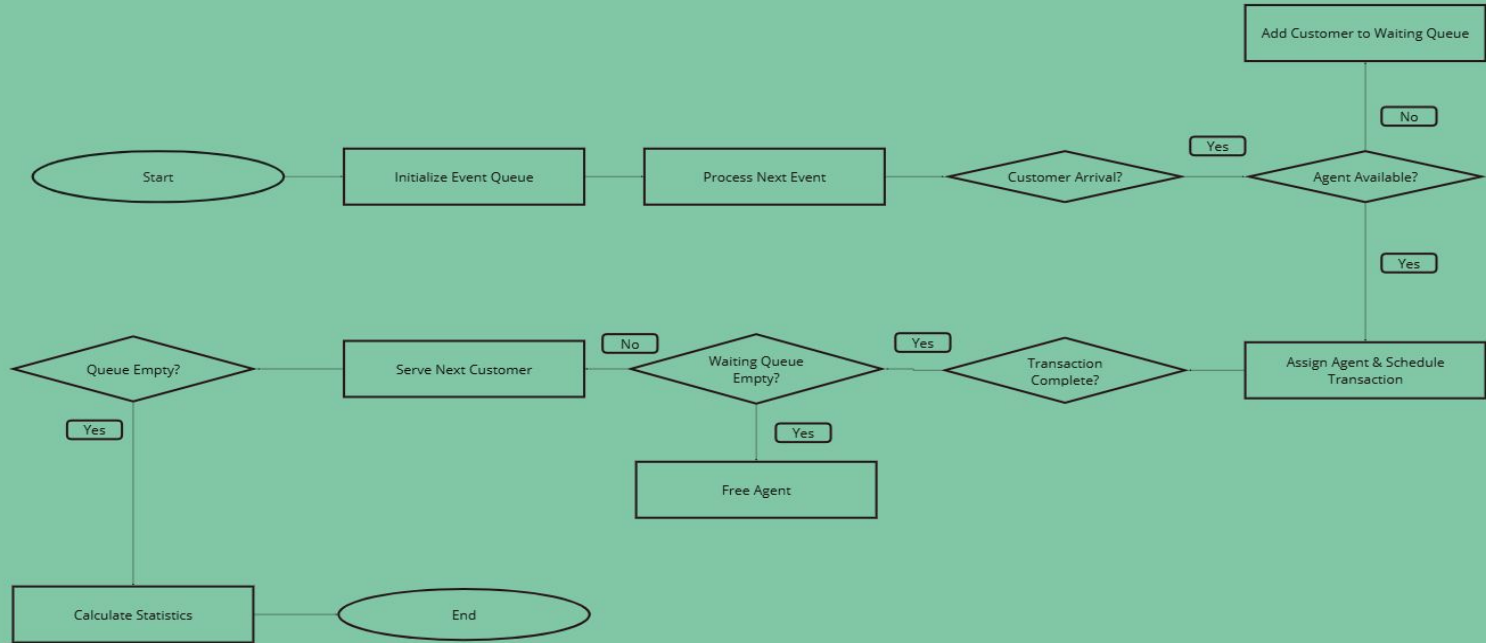


## 5.3 Flow Diagram: Simulation Workflow

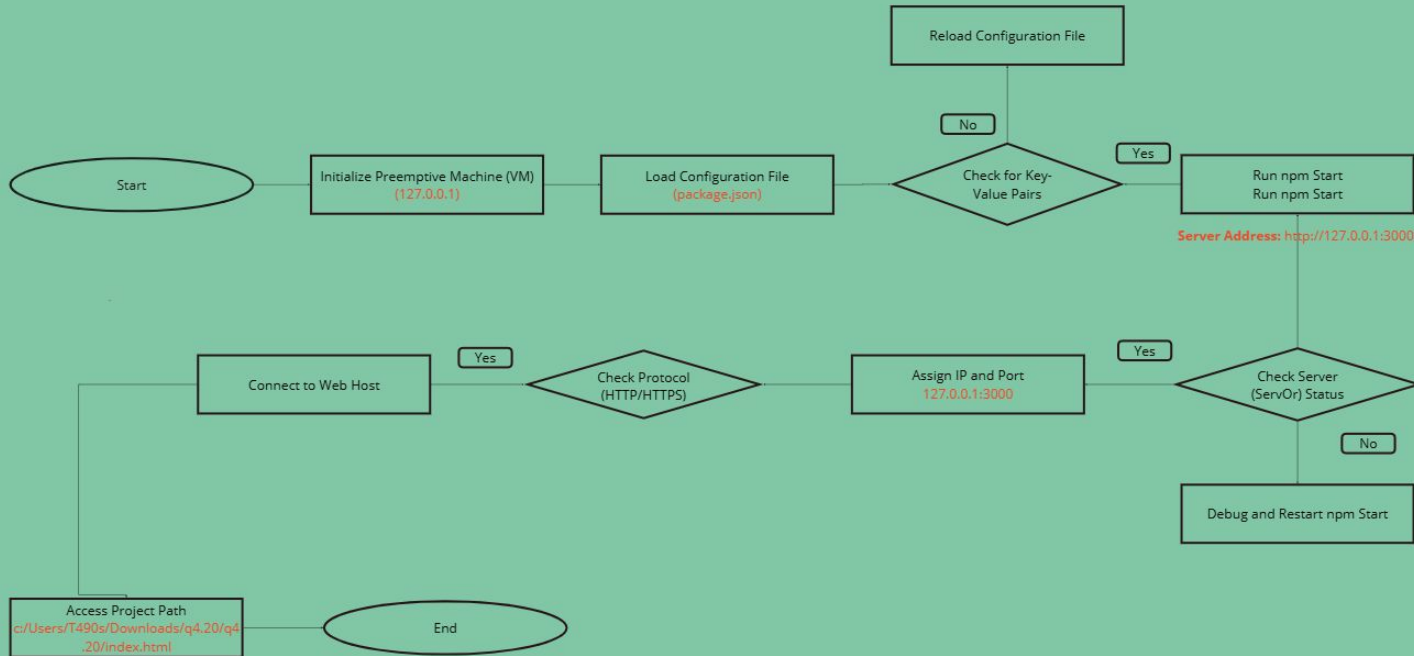
### Nodes & Interactions:

- Customers → Event Queue → Agents → Processing → Statistics
- **Event types:** Customer Arrival, Transaction Complete
- **Decision points:** Is an agent available?, Is waiting queue empty?
- Loops until the event queue is empty.

## 5.3 Flow Diagram: Simulation Workflow



## 5.4 Program Flow Diagram:





## 6. ALTERNATE APPROACHES:

### 6.1 Problem Context:

The problem involves simulating an **airline ticket counter** using **Discrete Event Simulation (DES)** to analyze customer service operations. Events like customer arrivals and transaction completions are processed in a **priority queue**, ensuring that events occur in the correct order. The simulation tracks customer wait times, agent availability, and queue length, updating the system as customers arrive, are served, or leave. The goal is to calculate the **average customer wait time** at the end of the simulation.

### 6.2 Why Explore Alternate Approaches?

- To find scalable, efficient, and real-time solutions for **Internet Application Development**.
- Explore methods beyond DES for better optimization.

### 6.3 Why AI-Driven Approach?

- **Real-Time Adaptability:** AI models can adapt to dynamic changes in customer behavior and system load.
- **Predictive Power:** Machine learning can predict wait times and optimize resource allocation in real-time.
- **Relevance to Internet Applications:** AI-driven solutions are widely used in modern web applications for tasks like recommendation engines, load balancing, and predictive analytics.

## 6.4 How It Works:

An AI-driven approach replaces the event-based simulation with a machine learning model that predicts customer wait times based on historical data. Instead of tracking individual events dynamically, the AI model takes inputs like arrival times, queue lengths, and agent availability to provide an instant prediction. This approach eliminates the need for a priority queue and event processing, making it faster and more efficient for real-time decision-making.

1. An AI API endpoint is defined.
2. The `fetch()` function is used to send a POST request.
3. The API response is converted to JSON.
4. The predicted wait time is returned.
5. The function is called, and the result is printed to the console.

## 6.5 BASIC LOGIC:

```
// Example: Simple AI-driven wait time prediction using a pre-trained model
async function predictWaitTime(customerData) {
  const modelEndpoint = "https://api.example.com/predict-wait-time";
  const response = await fetch(modelEndpoint, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(customerData),
  });
  const result = await response.json();
  return result.waitTime;
}

// Example usage
const customerData = {
  arrivalTime: 10,
  serviceTime: 3,
  numAgents: 2,
  queueLength: 5,
};

predictWaitTime(customerData).then(waitTime => {
  console.log("Predicted Wait Time:", waitTime);
});
```



## 6.6 Comparison Table:

Feature	Discrete Event Simulation (DES)	AI-Driven Approach
Real-Time Adaptability	Limited – processes events in fixed chronological order.	High – adapts to dynamic changes in real-time.
Predictive Power	None – relies on predefined event sequences.	High – predicts future wait times and optimizes resources.
Scalability	Scalable but computationally intensive for large systems.	Highly scalable for large and complex systems.
Implementation	Simple - Simulates events dynamically using a <b>priority queue</b>	Complex – Uses a <b>machine learning model</b> to predict outcomes instantly
Use Cases	Suitable for static, well-defined systems.	Ideal for dynamic, real-time systems (e.g., web apps).





## 7. Alternate Solution

Our project is built using HTML, CSS, and JavaScript for an Airline Ticket Counter Simulation. It uses event-driven simulation and handles user interactions entirely on the front end. However, if you wanted to implement the same project using other technologies, we could choose different programming languages and frameworks based on our needs.

A detailed comparison of alternative solutions with example codes is given below:

### Python (Flask + JavaScript)

### Python (Flask + JavaScript)

#### ✓ Why use Python?

Python is widely used for backend development because of its simplicity and powerful built-in libraries. With Flask (lightweight) or Django (full-featured), you can handle backend logic while keeping JavaScript for frontend interactions.

#### What is Flask (Lightweight Framework)?

Flask is a lightweight web framework that is simple and flexible. It only provides basic features, and if you need extra functionalities (like database, authentication, etc.), you have to add them manually.

#### Key Features of Flask

- ✓ Minimal and fast framework
- ✓ Best for **simple APIs** and small projects
- ✓ More **customization options**
- ✓ **Beginner-friendly** and easy to learn



## ⚡ Comparison with HTML, CSS, and JS Approach

Feature	HTML, CSS, JS (Current)	Python (Flask)
Type	Frontend only	Full Stack
Scalability	Limited	Better Scalability
Performance	Runs in browser	Faster-processing(Server-side)
Data Handling	No database support	Can use SQL, SQLite, MySQL
Complexity(Time and Space)	$O(n \log n) - O(n^2)$ and $O(n)$	$O(n \log n) - O(n)$ and $O(n)$

- Python (Flask) is better for large-scale applications because of its lower time complexity and better memory management.
- JavaScript (Frontend-only) struggles with large simulations due to higher event-handling complexity ( $O(n^2)$  in worst cases).



## 8. Future Improvements for the Simulation System

### *Why Improvements Are Necessary?*

- To make the system dynamic and engaging.
- To improve the user interface and accessibility.
- To simulate real-world scenarios more effectively.
- To track performance metrics for better insights.



## 8.1. Randomized Customer Arrivals

### Current Challenge:

- Customer arrival times are manually input and service times are fixed (e.g., 3 minutes), creating a rigid and unrealistic simulation.

### Improvement:

- **Introduce Probabilistic Models:**
  - Use **Poisson Distribution** to model the number of arrivals in a fixed time, **Exponential Distribution** models **time intervals** between customer arrivals
  - **Exponential or Normal Distribution** models **service times** for each customer, introducing variability.

### Benefits:

- Simulates the natural variability in both arrivals and service times.
- Provides more accurate insights into system performance under different conditions.



## 8.2. Real-Time Metrics and Reports

### Current Issue:

- Metrics like average wait time or agent utilization are only available after the simulation ends, limiting real-time insights.

### Improvement:

- Display **live statistics** during the simulation for better monitoring. Include metrics like **Number of customers in the waiting line, Average wait time at any moment, etc**
- Create a live **dashboard** that refreshes in real-time.

### Benefits:

- Enables immediate action if queue lengths or wait times become too high.
- Provides users with insights to optimize resource allocation during the simulation.



## 8.3. Queuing Discipline: Enhancing Customer Service Efficiency

### Current Issue:

- **First-Come-First-Serve (FCFS)** is the current queuing discipline.
  - Customers are processed in the order they arrive, which may lead to inefficient handling, especially when customers have varying service times.

### Improvements:

1. **Shortest Job Next (SJN):**
  - Prioritize customers with **shorter service times** to minimize overall waiting time.
2. **Priority Queue:**
  - Introduce a **priority system** based on criteria like **loyalty** or **urgency**, allowing certain customers to be served faster.

### Benefits:

- SJN prioritizes quicker transactions.
- Priority Queue serves high-value or urgent customers first.



## 8.4. Key Frameworks for Enhancement

### Event Simulation & Logic:

- **Node.js** : Server-side event processing and scalable simulations.

### Frontend & UI:

- **React.js/Flutter** :: Interactive UIs for real-time simulation visualization.

### Real-time Interaction:

- **Socket.io**: Real-time communication for live event updates and queue processing.

### Data Management & Analysis:

- **MongoDB**: Storing event data, customer queues, and simulation results.



Thank you!

