System Model

1. **Architecture & Components**

   Our system follows a **microservices** architecture, deployed via **Docker** and coordinated by **Docker Compose**. The key components are:

   - **Frontend (Browser-based UI):** A user-facing component that displays items, collects checkout data, and submits orders via **HTTP** to the Orchestrator.

   - **Orchestrator (Flask/REST):** Receives orders from the Frontend, assigns a unique OrderID, coordinates partial-order events (Transaction checks, Fraud checks, Suggestions generation) using **gRPC** calls to backend microservices.

   - **Transaction Verification, Fraud Detection, Suggestions:** Each is an independent microservice, exposing **gRPC** services on different ports. They may run concurrently and maintain internal **vector clocks** to track event ordering in a distributed environment.

   - **Order Queue:** Maintains a priority queue for orders. When an order is approved, the Orchestrator enqueues it in this service.

   - **Order Executor:** Multiple identical instances run the same code. They use a **Bully** leader election algorithm to ensure that only the elected leader dequeues and executes orders from the queue, providing mutual exclusion in processing.

2. **Connections Between Services**

   - **Frontend → Orchestrator:** REST/HTTP on port 8081 (mapped to 5000 internally).

   - **Orchestrator → Microservices (Transaction, Fraud, Suggestions):** gRPC calls, each microservice listening on a specific port (50052, 50051, 50053 respectively).

   - **Orchestrator → Order Queue:** Another gRPC call (port 50055) to enqueue orders.

   - **Order Executor → Order Queue:** The leader executor periodically polls the queue (gRPC) to dequeue orders. Executors also communicate among themselves for **leader election** messages (Election, Coordinator).

3. **Failure Modes**

- **Microservice Failure:** If a microservice (e.g., Fraud Detection) goes down, the Orchestrator's gRPC call to it may fail. The Orchestrator can handle timeouts or default to rejecting the order.

- **Leader Executor Failure:** If the current leader stops responding, other executor replicas detect the missing coordinator messages or failed RPC calls. They trigger a new Bully election and select a new leader.

- **Order Queue Failure:** If the queue is unavailable, enqueuing or dequeuing fails. The Orchestrator logs an error; the executors cannot retrieve orders until the queue recovers.

- **Network Partition:** If the executors cannot reach each other or the queue, they cannot coordinate effectively. Each might assume it's leader if it can't see a higher ID node. Once the partition is healed, only the highest ID node eventually remains leader (per the Bully algorithm).

- **Data Consistency:** Vector clocks ensure a well-defined partial order of events; in the worst case of network errors, some clock updates may be delayed, but once connectivity is restored, the Orchestrator merges all partial results or rejects incomplete data.

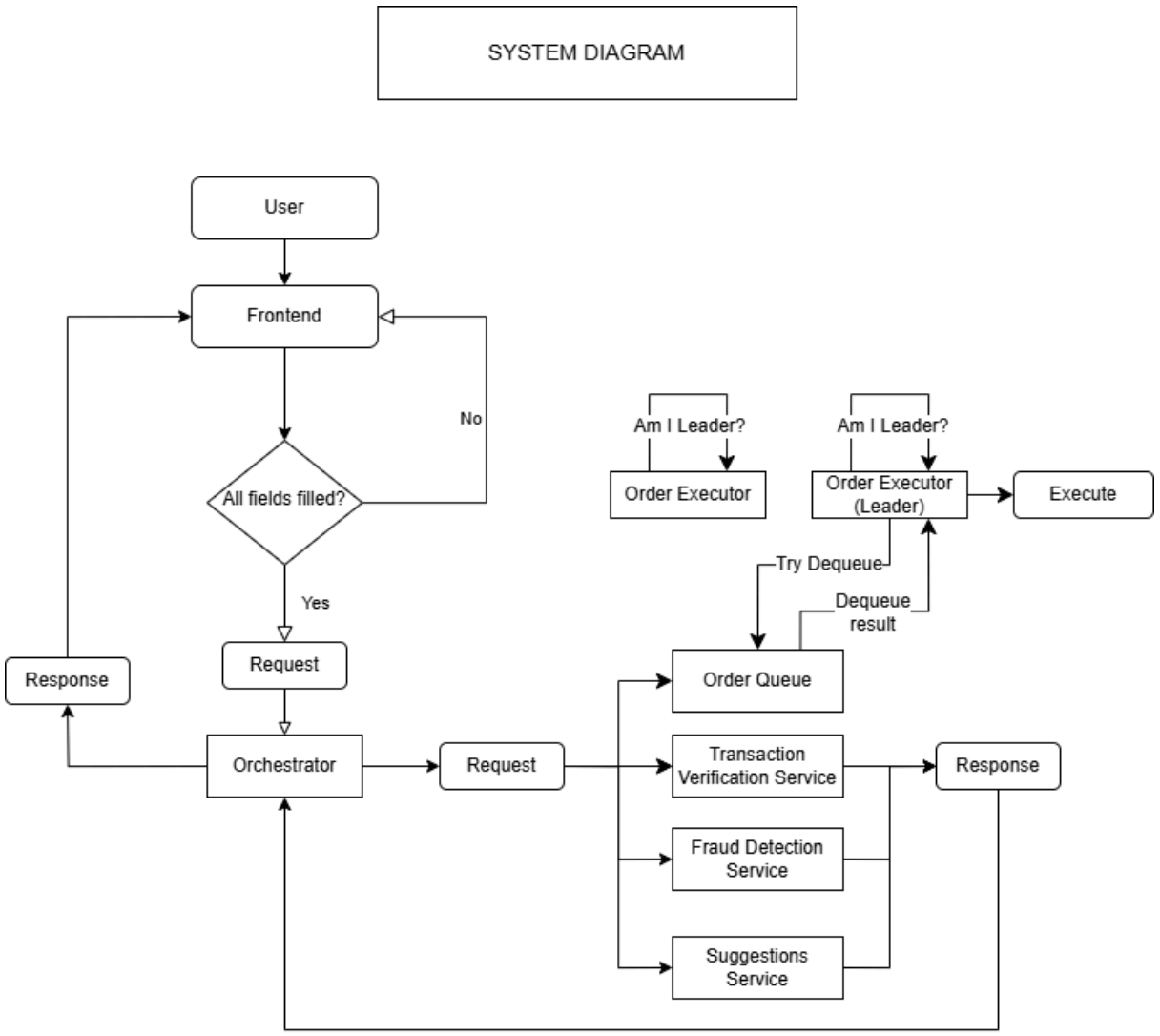
4. **Distributed Properties**

    - **Concurrency:** Multiple microservices handle events in parallel (Transaction checks, Fraud checks, Suggestions). Vector clocks maintain causality across these events.

    - **Scalability:** Adding more executor replicas or more instances of microservices can scale out the system. The highest ID replica among the executors is always the leader.

    - **Fault Tolerance:** If a leader fails, a new one is elected. If a microservice fails, orders may be rejected or queued until the service recovers, depending on the logic in the Orchestrator.

5. **Diagram**



A typical system diagram might show:

- The **frontend** calling the **Orchestrator** via REST.
- The **Orchestrator** calling **Transaction Verification**, **Fraud Detection**, **Suggestions** in parallel via gRPC.
- A separate **Order Queue** service, which stores approved orders in a priority queue.
- **Replicated Order Executors** connected both to each other (for leader election) and to the **Order Queue** (to dequeue orders).