Product Requirements Document (PRD)

1. Summary (Intent of Product)

The goal is to build a high-performance **mock trading engine** for a stock-market simulation event. The system will simulate real-world exchange behavior, allowing participants to send orders, observe market data streams, and experience realistic order matching, trade generation, and price movement. The engine must be fast, scalable, and support clean integration with data APIs, websockets, and orchestration tools.

---

2. Problem

Real financial exchanges involve sophisticated high-speed engines, complex market data flows, and distributed systems. Students and participants rarely get to experience or interact with such systems due to:

- Lack of accessible, high-performance mock trading infrastructure.
- Complexity in building event-driven and low-latency backend systems.
- Need for real-time market data simulation and order-matching.
- Difficulty in scaling multiple services (data, APIs, engine, messaging, caching, orchestration).

This simulation event requires an exchange-like environment that is simple to operate but realistic enough to allow participants to test trade execution logic.

---

3. Solution

We propose building a distributed **Mock Stock Exchange System** consisting of:

- A **C++ Trading Engine** performing ultra-fast order matching.
- A **Data Server** to provide mock market data feeds.
- **API/Websocket Server** for participants to place orders and stream updates.
- **Kafka** for event-driven data distribution.
- **Redis** for caching order books and maintaining fast lookup states.
- Containerized services orchestrated using Docker/Kubernetes.

This makes the system:

- Realistic: mimics real exchange components.
- Modular: each component is independent.
- Scalable: supports multiple participants.
- Safe: processes only mock data.

---

4. Features

Core Features

- Limit/Market Order Support (GFD, IOC, FOK)
- Real-time Matching Engine in C++
- Order Book Management with Redis caching
- Market Data Stream (tick-level)
- Websocket Push Updates for:
- Order confirmations
- Trade confirmations
- Market data

System Features

- Distributed microservices via containerized deployments
- Kafka-based message queues for event propagation
- System monitoring dashboards (optional)
- Fault-tolerant service structure

Master Architecture Document

1. APIs and Websocket Engines

REST APIs

1. **POST /order/place**
2. Submit a new order
3. Payload: { userId, symbol, side, quantity, price, type }
4. **GET /order/status/{orderId}**
5. Fetch order status
6. **GET /market/quote/{symbol}**
7. Fetch LTP, bid/ask, volume
8. **GET /market/orderbook/{symbol}**
9. Fetch full order book snapshot

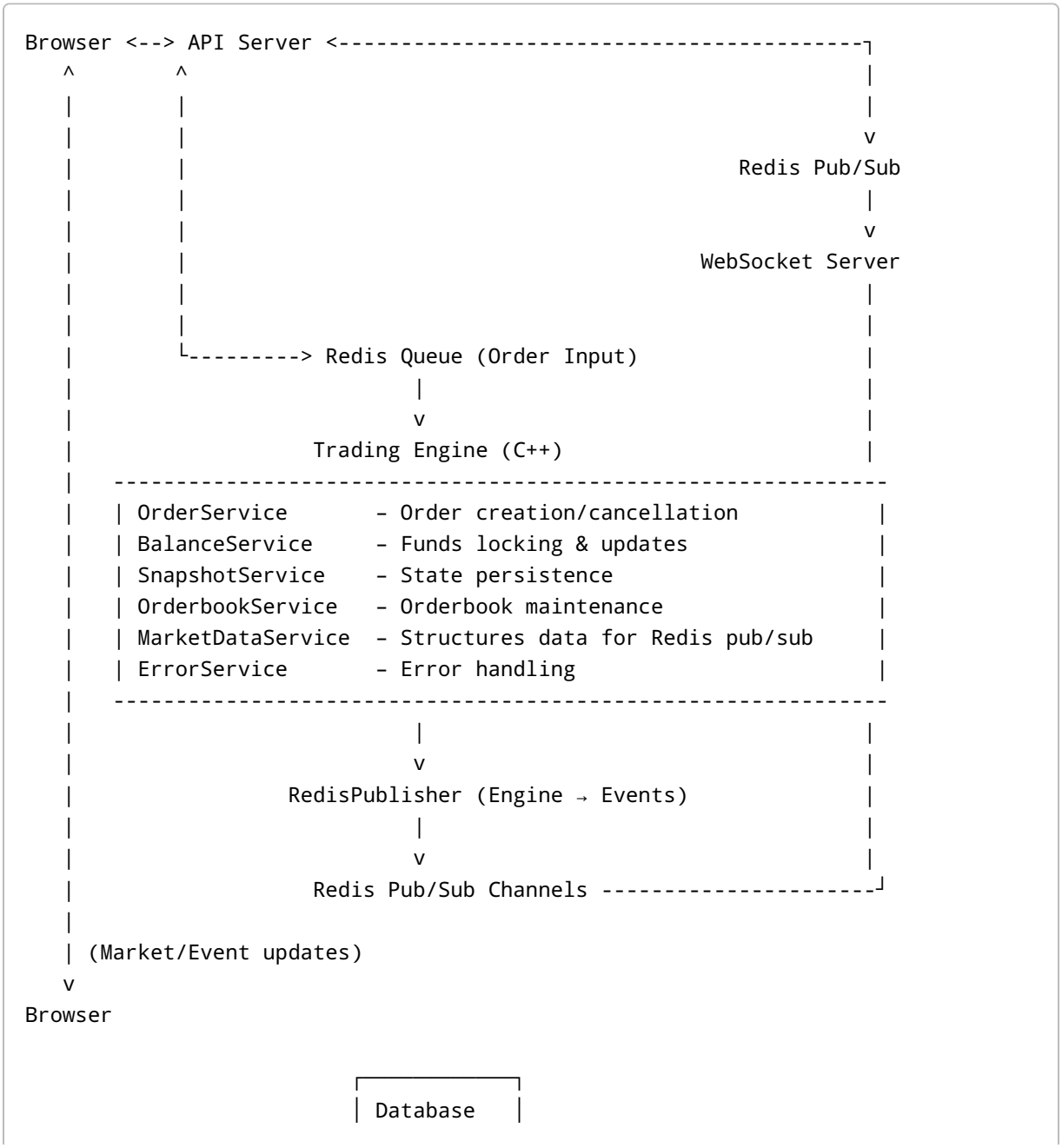
Websocket Channels

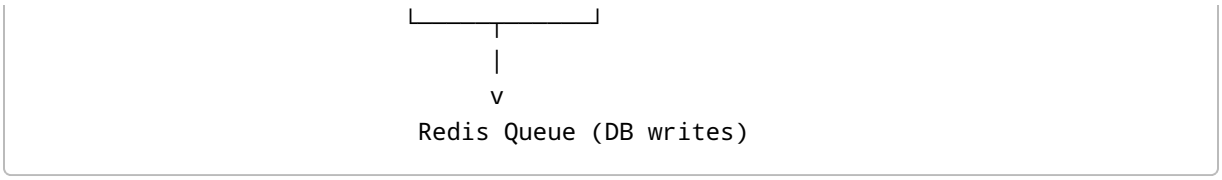
1. **/ws/marketdata** — Live price & depth stream
2. **/ws/orderupdates** — Live order status, trade confirmations
3. **/ws/exchange_events** — System events, logs (optional)

---

2. Low System Design

Updated System Architecture (Based on Provided Diagram)





Component Flow Based on Diagram

1. Browser

- Sends user actions → API server
- Receives push updates → WebSocket server

2. API Server

- Receives REST calls
- Pushes validated orders → Redis Queue
- Receives engine events via Redis Pub/Sub
- Responds back to browser

3. Trading Engine (C++)

- Consumes orders from Redis Queue
- Runs matching via internal services (OrderService, BalanceService, etc.)
- Publishes market data, order updates, and errors → RedisPublisher → Redis Pub/Sub
- Sends DB persistence events → Redis Queue for DB

4. WebSocket Server

- Subscribes to Redis Pub/Sub channels
- Streams:
- market data
- order updates
- error messages

5. Database Layer

- Listens to Redis Queue for persistence events
- Stores:
- trades
- order states
- snapshots

3. Tech Stack

Languages

- **C++** → Trading Engine, preferably API Server
- **Python/Go/Node.js** → Data Server (flexible)

Infrastructure

- **Kafka** → Event-stream backbone
- **Redis** → Low-latency caching layer
- **Docker / Kubernetes** → Orchestration & deployment
- **GitHub + CI/CD** → Build pipelines

---

4. Services Overview

i. Data Server

- Any language
- Generates market data → Kafka
- Pulls snapshots from Redis when needed

ii. API/Webhook Server

- Preferred C++ for tight Kafka/Redis integration
- Handles user traffic and websockets

iii. Trading Engine (C++)

- Core matching engine
- Very low latency
- Direct Redis + Kafka integration

iv. Kafka/Redis Images

- Use official Docker images
- Preconfigured topics and persistence

Orchestration

- All components containerized
- Deployed using **Docker Compose or Kubernetes**
- Separate network for:
- Engine cluster
- API cluster
- Data cluster
- Auto-restart policies
- Optional observability:
- Prometheus + Grafana
- Kafka UI
- Redis Insight
