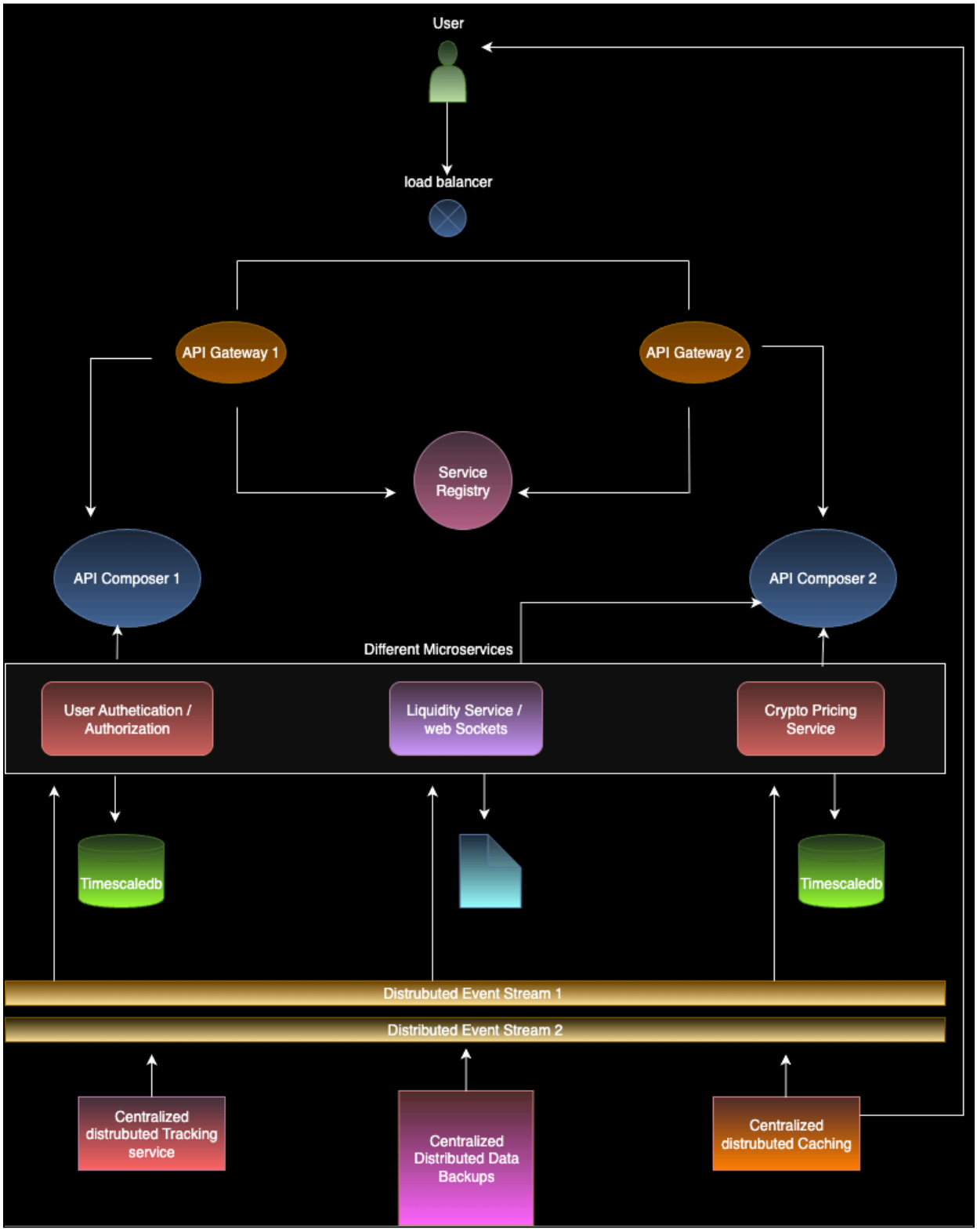
OPEN ENDED QUESTIONS

1) Critique the design you just built. What are its primary weaknesses or potential bottlenecks (API rate limits, network latency, single points of failure)

Ans -> My current system has a few constraints. The major one would be the non-availability of public crypto exchange APIs due to which I created a dummy infrastructure and have used functions to generate random, dummy prices for crypto currencies.
Assuming, we pick live, tick-by-tick from exchanges following are the weaknesses and solutions that would help make the system more robust and scalable.

Weakness	Description	Mitigation
1) API Rate Limits	Exchanges limit REST/WebSocket frequency	Use backoff strategies, token buckets, and Redis-based rate guards
2) Network Latency / Packet Loss	Especially critical for market orders	Co-locate with exchange regions (AWS, GCP) and use WebSockets > polling
3) Single Points of Failure	Like a single message queue or DB	Distributed event streams will handle single points of failure
4) WebSocket Interruptions	WebSocket drops can lead to state desync	Auto-reconnect + REST resync snapshot
5) DB write contention	Heavy writes from order tracking	Using Timescaledb compression and indexing
6) Time Inconsistency	Latency between order trigger and placement	Timestamp everything at origin and synchronize using NTP (will be handled using Timescaledb)
7) Data Loss on Crashes	Market/order updates lost on service crash	Persist state snapshots frequently, replay from queue logs
8) Constant API hits	Will increase data retrieval time	Caching systems will help users access commonly asked data points.

My solution would abide by the following diagram ->



2) How will you evolve this prototype into production grade, highly available system? Briefly describe the architecture, frameworks or technologies you would use.

Ans -> Components Overview -

1) Frontend / UI / CLI :-
- User-facing interface to place orders and track statuses.

2) API Gateway / Loadbalancer :-
- Routes requests to appropriate services
- Provides rate limiting, auth and SSL termination

3) Order Service :-
- Handles placing, cancelling and modifying orders
- Integrates with REST API endpoints of exchanges

4) Order Status Tracker :-
- Subscribes to websocket streams for real time order updates
- Handles auto-reconnect and falls back to REST.

5) Market Data Ingest :-
- Streams and polls order books, trades, funding rates, etc

6) Message Queue (Kafka / RabbitMQ / Redis Streams)
- Buffers all incoming market and order events.
- Enables retry mechanisms and asynchronous processing.

7) Order Processor :-
- Consumes messages from the queue.
- Places orders, updates DB, and handles exchange communication.

8) Exchange API Layer :-
- REST + WebSocket clients for each exchange.
- Load-balanced and retry-aware.

9) TimescaleDB (PostgreSQL) :-
- Time-series optimized DB for order logs, snapshots, and market data.

10) Monitoring and Observability :-
- Prometheus + Grafana for metrics.
- ELK stack for logs.

Key Production Features:

- Idempotent order IDs to avoid duplication.
- Exponential backoff with jitter for all retries.
- WebSocket heartbeat/ping for connection liveness.
- State snapshotting for recovery.
- Circuit breaker for unstable endpoints.
- REST polling fallback for WebSocket disconnects.
- Async task queues for workload balancing.
- Cold start bootstrapping from latest DB snapshot.
- Auto-scaling based on latency and backlog.

3) Describe a robust system for handling API errors from exchanges, sudden exchange downtime or websocket connection interruptions.

Ans -> The aim is to ensure reliability, data consistency, and continued functionality during network or service instability.

1. **Retry Logic with Exponential Backoff + Jitter -**
   - Gradually increases wait time between retries.
   - Jitter prevents simultaneous retry storms.

Sample Logic:

```
for attempt in range(MAX_RETRIES):
    try:
        return make_api_call()
    except APIError:
        time.sleep((2 ** attempt) + random.uniform(0, 1))
```

2. **Circuit Breaker Pattern -**
   - Monitors for consecutive failures.
   - Temporarily stops calls to unstable APIs.
   - Redirects to fallback systems (cache/synthetic data).

3. **WebSocket Resilience**
   - Use heartbeat ping/pong messages.
   - Auto-reconnect logic with re-subscriptions.
   - Sync state with REST APIs after reconnection.

4. **REST API Fallback -**
   - If WebSocket fails, use REST polling.
   - Ensures continuity with reduced latency.

15. **Idempotent Order Operations -**
   - Generate unique client-side order IDs.
   - Ensures safe retries without duplicate execution.

6. **Message Queuing System -**
   - Tools: Kafka, Redis Streams, RabbitMQ.
   - Decouples API data from consumers.
   - Buffers during downtimes for later processing.

7. **Atomic Database Transactions** -
   - Use transactional DB writes (e.g. PostgreSQL).
   - Ensure order states, fills, balances are atomically saved.
   - Periodic snapshots of system state.

8. **Monitoring, Alerts & Self-Healing -**
   - Track API errors, latency, missed heartbeats.
   - Auto-restart modules on crash.
   - Send alerts via Slack, PagerDuty, etc.

9. **Rate Limit Management** -
   - Read API headers (e.g. X-RateLimit-Remaining).
   - Throttle low-priority requests.
   - Prioritize critical order/trade submissions.

10. **Redundancy & Failover -**
   - Use load balancers and redundant services.
   - Multi-region deployment for critical components.

11. **Graceful Shutdown and Recovery** -
   - Save in-memory state before shutdown.
   - Rehydrate state from disk/message logs on restart.

This system architecture provides:

1. High availability
2. Real-time error recovery
3. Resilient data pipelines
4. Strong monitoring and recovery mechanisms

Common Tools -

Component	Technology
Retry Logic	`tenacity`, custom logic
WebSocket	`aiohttp`, `websockets`
Message Queue	Kafka / Redis Streams
Database	TimescaleDB/PostgreSQL
Monitoring	Prometheus + Grafana
Alerts	PagerDuty, Slack

It is a foundational design for production-grade financial systems that need to withstand exchange-related disruptions.