2.2 Bottleneck Identification

Answer:

The system faces five potential bottlenecks:

1. **Database queries**
    - Redis may become overloaded by frequent feature lookups.
    - TimescaleDB can slow down due to large scans of time-series data.
    - ClickHouse may struggle with expensive ad-hoc analytics queries.

      Optimization: Use pipelined queries and sharding in Redis, partition and rollups in TimescaleDB, and materialized views in ClickHouse.
2. **API rate limits**
    - Hyperliquid APIs may reject orders if too many requests are sent at once.
    - Third-party APIs for sentiment or economic data can also throttle requests.

      Optimization: Apply token-bucket rate limiting, use idempotency keys, batch operations, and introduce caching with short TTLs to reduce redundant requests.
3. **Network latency**
    - Cross-region traffic and repeated serialization can slow down data delivery.

      Optimization: Co-locate latency-sensitive components, use long-lived WebSocket connections, and compress messages using binary formats.
4. **Computational overhead**
    - Stream processors may waste time recomputing windows or parsing JSON.
    - Agents can slow down due to heavy inference or noisy signals.

      Optimization: Use incremental window updates, vectorized math (NumPy/Numba), batch inference, and debounce agent decisions.
5. **Concurrency issues**

- Multiple agents could race to trade the same symbol, or the router could become a bottleneck.

  Optimization: Use a Redis “blackboard” for intents, enforce a central risk gate for final approval, and design the order router as stateless replicas with shared idempotency storage.

2.3 Data Pipeline Design

Answer:

1. **Real-time price feeds**
   - Hyperliquid WebSocket → normalize messages → event bus → stream processors → Redis feature store → agents.
   - This design ensures sub-500 ms latency because all steps are in-memory and event-driven.
2. **Historical data**
   - Subgraph/indexing providers (e.g., Envio/Goldsky) are used to fetch on-chain history.
   - Results are stored in TimescaleDB (time-series queries) and ClickHouse (analytics).
   - Cached in Redis for frequently requested queries.
3. **Extensibility**
   - New data sources (e.g., Twitter sentiment) plug in through a connector interface (`connect()`, `normalize()`, `emit()`).
   - All connectors output standardized messages into the event bus, so the core system does not need changes.
4. **Data freshness vs. consistency**
   - For trading, prioritize freshness: agents act on the latest available data, even if slightly inconsistent.
   - For reporting, prioritize consistency: use reconciled subgraph data for audits.

- Features include timestamps; agents enforce staleness rules (e.g., price <500 ms old, sentiment <5 minutes).

2.4 Multi-Agent State Management

Answer:

- **Sharing information without races**: Agents publish their latest trading intent to a shared Redis “blackboard” (`intent:BTC = {agent, side, size, ts}`), which others can read. This avoids direct overwrites.
- **Preventing conflicting orders**: All intents pass through a central risk engine. It enforces exposure, leverage, and rate limits before forwarding approved intents to the order router. The router uses idempotency keys to avoid duplicate orders.
- **Maintaining audit trails**: Every step (intent creation, approval, submission, fills) is logged as an append-only event in TimescaleDB for operations and ClickHouse for analytics. Each entry includes IDs and timestamps to enable full replay.
- **Handling crashes**:
  - Agents send heartbeats; if one fails, the risk engine cancels its pending intents and the router cleans up orders.
  - Routers are stateless replicas, so they can restart and replay safely without duplicates.
  - Ingestion services can replay missed events from the event bus.

Summary: Redis provides fast context sharing, the risk engine ensures safe execution, audit trails guarantee traceability, and heartbeats with idempotency provide resilience.
