The following document is a System Design Document showcasing the software design for a trading platform, done by Joe Archondis

Trading Platform

System Design Document

Version 2.0

09/06/2025

Table of Contents

1. Introduction 5

2. System Requirements 6

2.1 Functional Requirements 6
2.1.1 User Registration and Authentication 6
2.1.2 Market Data Processing 6
2.1.3 Order Routing and Management 6
2.1.4 Trade Execution 7
2.1.5 Portfolio Management 7
2.1.6 Risk Management 7
2.1.7 Notifications and Alerts 7
2.2 Non-Functional Requirements 8
2.2.1 Performance 8
2.2.2 Scalability & Reliability 9
2.2.3 Analytics 10
2.2.4 Compliance 10

3. Database Design 11

3.1 Database Diagram 11
3.2 Micro-services Breakdown 12
3.2.1 User Data (User Management Service) 12
3.2.2 Market Data (Market Data Service) 12
3.2.3 Order Data (Order Management System & Trade Execution Engine) 13
3.2.4 User Portfolio & Transaction History (Portfolio Management Service) 13
3.2.5 Audit Logs & Regulatory Data (Compliance Service) 13
3.2.6 Additional Considerations 13

4. API Design 14

4.1 User Management APIs 14
4.2 Market Data APIs 14
4.3 Order Management APIs 14
4.4 Portfolio Management APIs 15
4.5 Notifications APIs 15
4.6 Security Considerations 15

5. High-level Design 16

5.1 Client (Web/Mobile) 16
5.2 Load balancer 17
5.3 API Gateway 17
5.4 Authentication Service 18
5.5 Order Service 18
5.6 Trade Execution Service 19
5.7 Market Data Service 19
5.8 Risk Management Service 20
5.9 Compliance Service 20
5.10 Notification Service 21
5.11 Databases Choice 21

6. Request Flows 22

6.1 Order Placement Flow 22

7. Detailed Component Design 23

7.1 Order Management System (OMS) 23
7.2 Market Data Service 24
7.3 Trade Execution Engine (TEE) 25

8. Trade offs/Tech choices 26

9. Failure Scenarios/Bottlenecks 27

10. Future Improvements 28

Appendix A: Record of Changes 29

Appendix B: Acronyms 30

Appendix C: Glossary 31

Appendix D: Referenced Documents 32

List of Figures

List of Tables

Table 1 - Record of Changes 26

Table 2 - Acronyms 27

Table 3 - Glossary 28

Table 4 - Referenced Documents 29

1. Introduction

This document outlines the system design and architecture of a high-frequency trading (HFT) platform tailored for the cryptocurrencies market.

The platform is engineered to handle massive volumes of data, analyze market conditions in real-time, and execute trades within microseconds to capitalize on fleeting market inefficiencies.

This document will present functional and non functional requirements, as well as a comprehensive view of the system's architecture, encompassing network design, database design, data processing pipelines and technical requirements.

Key objectives include:

- ➔ **Speed**: Leveraging advanced networking and optimized data structures to achieve ultra-low-latency performance.
- ➔ **Scalability**: Building a system capable of scaling to accommodate growing market demands and increased trading volumes.
- ➔ **Resilience**: Ensuring continuous operation through redundancy and failover mechanisms in the face of unpredictable market or infrastructure failures.
- ➔ **Security**: Protecting sensitive data and operations within a highly secure framework, adhering to best practices for financial systems.
- ➔ **Regulatory Compliance**: Designing the platform with flexibility to adapt to the regulatory requirements of various jurisdictions

2. System Requirements

2.1 Functional Requirements

2.1.1 User Registration and Authentication

- Implement strong password hashing algorithms (e.g. bcrypt) for secure password storage.
- Integrate with social login providers (e.g., Google, Facebook) for a convenient user experience (optional).

2.1.2 Market Data Processing

Traders should be able to preview aggregated and detailed market data from various markets.

- The system should stream live market data (Order Book, Ladder)
- The system should aggregate data (Synthetic order book, cross order book, etc.)
- Allow users to filter and customize market data views (watchlists, liquidity providers, exchanges).
- Support Historical data for analysis and back-testing
- Integrate with charting libraries for advanced technical analysis tools.
- Display news feeds and economic data relevant to specific cryptocurrencies. (optional) .

2.1.3 Order Routing and Management

Traders should be able to place buy and sell orders for various instruments

- Implement order types. Based on your position as a Market Taker or Market Maker, you might want to prioritize certain order types over others.
    - Basic Order types: Market orders, Limit orders, RFQ
    - Market Taker: Fill-Or-Kill, Immediate or Cancel, Smart Order Routing
    - Market Maker: Good Till Canceled, stop-loss orders
    - Trailing stop-loss and iceberg orders (for sophisticated traders)
- Orders must be routed to the appropriate exchanges or liquidity providers based on market conditions & user specification

2.1.4 Trade Execution

Trades should be placed and executed within a low-latency environment, while still ensuring high reliability. Trades should also have a detailed report for regulation compliance reports and back testing.

- Ensure fast execution of orders with low-latency in response to market conditions.
- Confirm orders and trade completion, showing detailed trade execution reports.

2.1.5 Portfolio Management

Traders should be able to preview Portfolio detailed data and reports, and have management tools at their disposal.

- Integrate with Market Data feeds to provide real-time portfolio valuation.
- Allow users to download transaction history and performance data in various formats (e.g., CSV, PDF).
- Offer basic portfolio re-balancing tools and investment strategy suggestions.

2.1.6 Risk Management

Traders should be able to implement risk management strategies

- Implement real-time risk management tools, including margin checks, exposure limits, and stop-loss mechanisms.
- Provide alerts for potential risks and trading limits being breached.

2.1.7 Notifications and Alerts

- Send notifications for order status (filled, partially filled, cancelled), price alerts, and risk alerts.
- Support both real-time alerts and periodic notifications (e.g., portfolio value changes).
- Allow users to customize notification preferences (e.g., email, SMS, in-app alerts) for different events.

2.2 Non-Functional Requirements

2.2.1 Performance

Latency Targets for Order Processing

For high-frequency trading (HFT) and ultra-low latency systems:

- **Sub-millisecond execution**: Ideally, **under 100 microseconds** for order submission and acknowledgment.
- **Tick-to-trade latency**: Below **500 microseconds** from receiving market data to executing an order.
- **Round-trip latency**: From order submission to confirmation, usually within **1 millisecond** or less.

For institutional trading and retail platforms:

- **Execution time within 1-10 milliseconds** for limit orders.
- **Market orders processed within 5-50 milliseconds**, depending on order size and routing.

Latency Targets for Market Data Updates

- **Exchange to trading system data propagation**: Ideally **sub-100 microseconds**.
- **Internal processing of market data**: Should be **under 500 microseconds** for HFT strategies.
- **Order book updates**: Visible within **1 millisecond** for HFT firms.
- **Retail platforms**: Market data updates within **10-100 milliseconds**, optimized for user experience rather than speed.
- Implement performance monitoring tools, logging and dashboards for continuous system optimization. (ELK stack)

Other Considerations

- **Network latency**: Ensuring proximity to exchanges and colocation services helps minimize delays.
- **Hardware and software optimizations**: FPGA acceleration, kernel bypass, and optimized algorithms can further improve performance.
- **Regulatory constraints**: Some markets impose latency floors or restrictions.

2.2.2 Scalability & Reliability

- Design the system using message queues and asynchronous processing to handle high volumes of concurrent users & to be able to process multiple streams of market data. (look for multi-threaded environments, such as ones offered by C++)
- Utilize cloud-based infrastructure with auto-scaling capabilities to accommodate spikes in trading activity. (consider AWS)
- Consider compartmentalizing your infrastructure. A good strategy for Trading systems is having the Market Data Processing handled in C++, and OTC trading handled in Rust. This ensures **optimized performance for each component** .

Key Scalability Requirements

➔ **Transaction Throughput:**

- Ability to process **X orders per second** at peak market activity.
- Support for **burst loads** during volatile periods (e.g., news releases).

➔ **Market Data Handling:**

- Capacity to ingest and process **Y market updates per second** from multiple exchanges.
- Ensure minimal impact on order execution latency during data spikes.

➔ **User Concurrency:**

- Support for **Z simultaneous users** without degradation in response time.
- Maintain performance with **increased connections** across multiple geographic locations.

➔ **Infrastructure Elasticity:**

- Dynamic scaling of compute and network resources based on trading activity.
- Load balancing across multiple servers and data centers.

➔ **Database Scalability:**

- Efficient handling of **growing trade history**, market data, and user transactions.

- Optimized query performance for real-time analytics and risk assessment.

➔ **Failover & Redundancy:**

- **No single point of failure**—automatic failover mechanisms to backup systems. (example: If a trade fails, try it again before giving up)
- Distributed architecture to maintain uptime during outages.

➔ **Geographic Scalability:**

- Ability to operate seamlessly across **multiple regions and exchanges**.
- Latency-optimized connections for colocated trading and cloud-based execution.

2.2.3 Analytics

➔ **Real-time metrics** like total orders executed, asset volatility, and exposure must be tracked for fast retrieval and analysis.

➔ **System logs and health monitoring** must be continuously captured and reviewed to ensure operational stability. (consider ELK stack)

➔ **High-frequency analytics** will be stored in a key-value database (e.g. **DynamoDB** from **AWS)** and aggregated daily to facilitate comprehensive, in-depth analysis.

By elaborating on these requirements, we provide a clearer picture of the functionalities and performance expectations for the trading platform. This detailed breakdown helps guide the design and development process to ensure a user-friendly, secure, and reliable trading experience.

2.2.4 Compliance

➔ The system must comply with global financial regulations (e.g., SEC, MiFID II, CFTC)

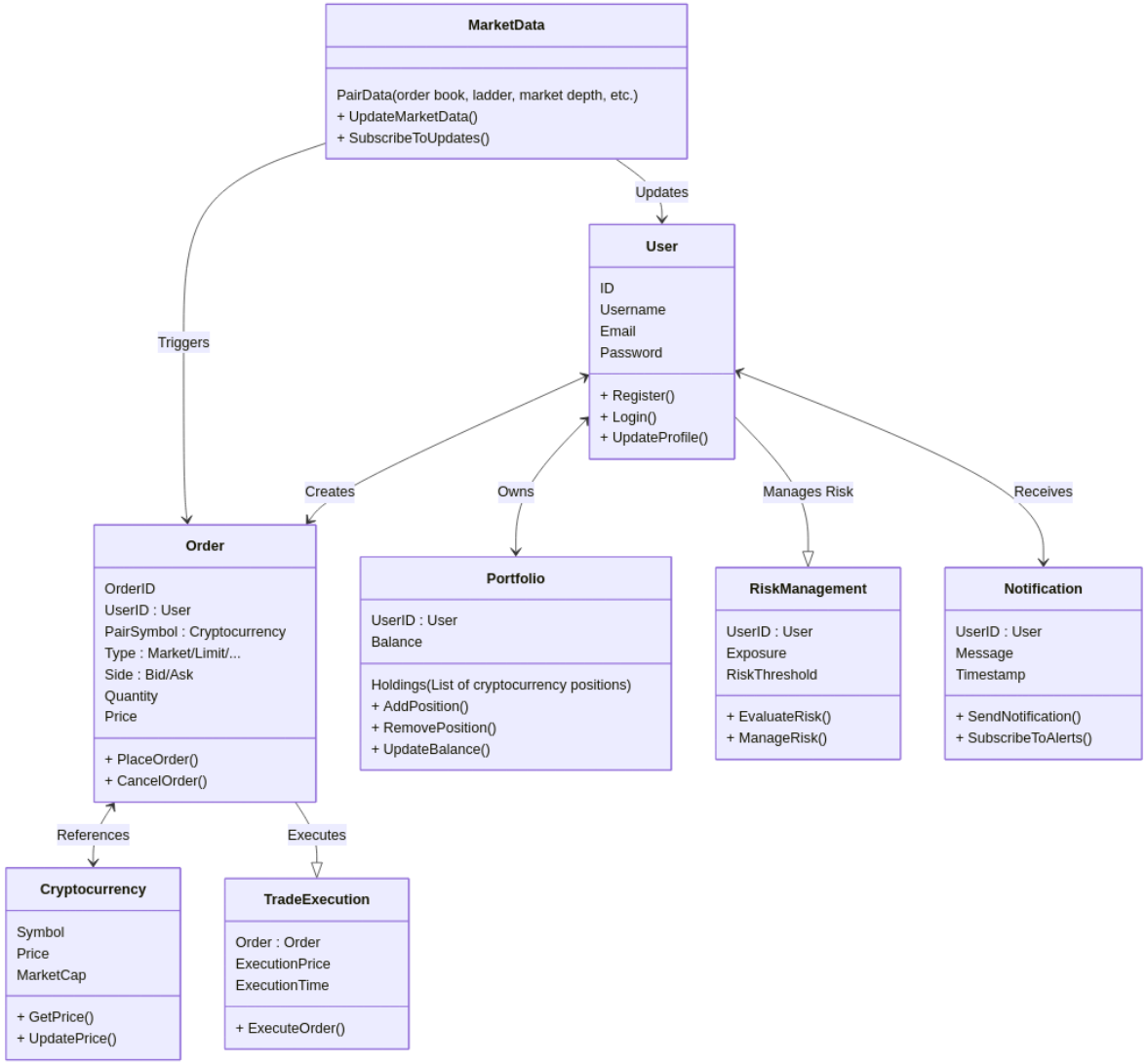
➔ Integrate with KYC/AML verification services to comply with anti-money laundering regulations

➔ Maintain detailed audit logs of all user activities and transactions for regulatory reporting purposes

➔ Provide trade reports for tax filings, audit trails, and compliance monitoring

3. Database Design

3.1 Database Diagram



3.2 Micro-services Breakdown

The CAP theorem states that in a distributed system, you can only have at most two of the following properties:

- Consistency: Every read reflects the latest write operation.
- Availability: Every request receives a (non-error) response, even if it's outdated data.

- Partition Tolerance: The system continues to operate even when network partitions occur

Here's a breakdown of suitable databases for different types of data in the inventory trading platform, considering the CAP theorem:

3.2.1 User Data (User Management Service)

- **Database Type:** Time-series Database (e.g. InfluxDB, TimescaleDB) or NoSQL Database (e.g. Elasticsearch, Cassandra)
- **Reasoning:** Time-series data (crypto prices, order book changes) requires fast writes and retrieval based on timestamps.
- **CAP Theorem:** Availability Focused - Real-time market data updates are crucial. Short-term inconsistencies can be tolerated for faster updates. Time-series databases are optimized for high write throughput and efficient time-based queries. Cassandra offers eventual consistency, ensuring data converges over time.

3.2.2 Market Data (Market Data Service)

- **Database Type:** Time-series Database (e.g. InfluxDB, TimescaleDB) or NoSQL Database (e.g. Elasticsearch, Cassandra)
- **Reasoning:** Time-series data (crypto prices, order book changes) requires fast writes and retrieval based on timestamps.
- **CAP Theorem:** Availability Focused - Real-time market data updates are crucial. Short-term inconsistencies can be tolerated for faster updates. Time-series databases are optimized for high write throughput and efficient time-based queries. Cassandra offers eventual consistency, ensuring data converges over time.

3.2.3 Order Data (Order Management System & Trade Execution Engine)

- **Database Type:** High-performance Key-Value Store (e.g. Redis) or In-Memory Database (e.g. Apache Ignite)
- **Reasoning:** Order data requires fast read/write access for order placement, execution, and management.

- **CAP Theorem:** Availability Focused - Order processing needs high responsiveness. Short-term inconsistencies can be resolved later through reconciliation processes. High-performance key-value stores offer fast access and scalability for order data.

3.2.4 User Portfolio & Transaction History (Portfolio Management Service)

- **Database Type:** SQL Database (PostgreSQL) or noSQL Database (e.g. MongoDB)
- **Reasoning:** Portfolio data requires consistency and efficient retrieval of user holdings and historical transactions.
- **CAP Theorem:** Balanced - Portfolio data is critical for users and requires consistency with a reasonable level of availability. SQL databases with strong consistency guarantees or noSQL databases with eventual consistency models can be considered based on specific needs.

3.2.5 Audit Logs & Regulatory Data (Compliance Service)

- **Database Type:** SQL Database (PostgreSQL)
- **Reasoning:** Audit logs and regulatory data require high durability and accurate record keeping.
- **CAP Theorem:** Consistency Focused - Regulatory compliance demands a strong focus on data integrity and retrievability. SQL databases with ACID transactions ensure data consistency for auditing purposes.

3.2.6 Additional Considerations

- **Hybrid Approach:** You may consider a hybrid approach where different database types are used for different functionalities based on the CAP theorem trade-offs.
- **Data Replication:** Data replication across geographically distributed servers can improve availability and fault tolerance.

4. API Design

This section outlines the API interface design and will break down the set of APIs needed to facilitate communication between different components and external applications.

4.1 User Management APIs

- Register User: Allows users to create new accounts with username, password, and other relevant information.
- Login User: Validates user credentials and provides authentication tokens for accessing other APIs.
- Get User Profile: Retrieves user information like account details, preferences, and settings.
- Update User Profile: Enables users to update their profile information.

4.2 Market Data APIs

- Request For Quotes: Provides real-time market data for specific crypto, including price, volume, and change.
- Get Risk Limits: Provides Risk Limit of cryptocurrencies
- Get Historical Data: Retrieves historical price data for a specified period.
- Get Order Book: Retrieves the current order book for a specific cryptocurrency, showing bids and asks orders at different price levels.
- Subscribe to Market Data: Allows users to subscribe to real-time updates for specific cryptocurrencies

4.3 Order Management APIs

- Place Order: Submits an order to buy or sell, specifying order type (market, limit, stop-loss, etc.), quantity, and price.
- Cancel Order: Allows users to cancel an existing order before it's filled.
- Get Order Status: Retrieves the current status of an order (pending, filled, cancelled).
- Get Order History: Provides a list of past orders placed by the user.

4.4 Portfolio Management APIs

- Get Portfolio Positions: Retrieves user's current holdings and their corresponding quantities and market values.
- Get Portfolio Performance: Provides performance metrics like total return, realized and unrealized P/L.
- Download Transaction History: Allows users to download a history of their transactions in a specified format.

4.5 Notifications APIs

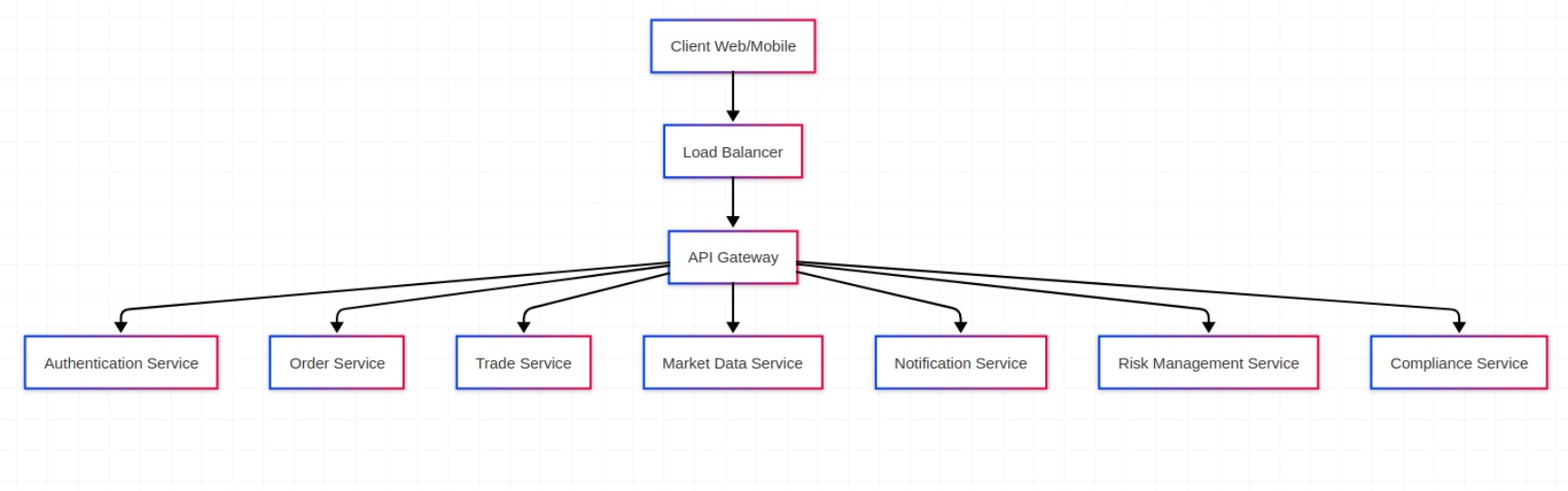
- Subscribe to Notifications: Enables users to subscribe to real-time notifications for order updates, price alerts, and market events.
- Send Notification: (Internal API) Used by other components to trigger notifications based on specific events (e.g., order execution, price threshold reached).

4.6 Security Considerations

- All APIs should be authenticated using secure tokens obtained through user login.
- Implement rate limiting to prevent abuse and ensure system stability under high load.
- Validate and sanitize user input to prevent security vulnerabilities like injection attacks.

5. High-level Design

This section identifies and breaks down key components needed for a complete end-to-end solution



5.1 Client (Web/Mobile)

- **Functionality**: The **Client** is the user interface that allows traders to interact with the stock trading system. It provides features such as placing orders, viewing portfolio and market data, receiving notifications, and managing their profile.
- **Responsibilities**:
  - Submit orders (buy/sell) and interact with market data.
  - Display real-time updates, including order status, trade execution, and market changes.
  - Display notifications and alerts (e.g., trade confirmation, risk alerts).
- **Data Flow**: Sends API requests to the **API Gateway**, receives responses containing market data, order details, and notifications.

5.2 Load balancer

- **Functionality**: The **Load Balancer** distributes incoming requests from clients across multiple backend servers to ensure availability, reliability, and efficient resource utilization.
- **Responsibilities**:
    - Ensure high availability and fault tolerance by distributing traffic evenly across multiple instances of backend services.
    - Handle sudden traffic spikes and ensure no single server gets overloaded.
- **Data Flow**: Receives client requests and forwards them to the **API Gateway**, optimizing traffic management.

5.3 API Gateway

Functionality: The **API Gateway** serves as the entry point for all client requests. It acts as a **reverse proxy**, routing requests to the appropriate backend services based on the request type.

- **Responsibilities**:
    - Handle authentication (verify JWT tokens) and forward user requests to relevant services.
    - Aggregate data from various services (e.g., user orders, trade executions) and send back unified responses.
    - Enforce security policies, such as rate-limiting and access control.
- **Data Flow**: Forwards requests from the **Client** to backend services like **Authentication Service**, **Order Service**, **Market Data Service**, etc.

5.4 Authentication Service

- **Functionality:** The **Authentication Service** is responsible for verifying users' identities and managing their session using JWT tokens.
- **Responsibilities**:
  - Authenticate users based on their login credentials.
  - Generate and verify **JWT tokens** for session management.
  - Handle user registration and password recovery.
- **Data Flow**: Receives login requests from the **API Gateway**, validates user credentials with **PostgreSQL** data, and returns a JWT token to the **Client**.

5.5 Order Service

- **Functionality:** The **Order Service** handles user orders, including creation, modification, and cancellation.
- **Responsibilities**:
  - Accept bid/ask orders from users and store them in the **PostgreSQL Orders Database**.
  - Validate orders and ensure they meet requirements (e.g. order type, price).
  - Update order status based on execution or cancellation.
- **Data Flow**: Receives order requests from the **API Gateway**, updates the **PostgreSQL Orders Database**, and notifies users about order status changes.

5.6 Trade Execution Service

- **Functionality:** The **Trade Service** is responsible for executing trades, including order matching, confirmation, and trade execution.
- **Responsibilities**:
    - Match buy and sell orders based on price-time priority and execute trades.
    - Confirm trade execution and store trade details in the **PostgreSQL Trades Database**.
    - Communicate with **Market Data Service** to check for the best available prices.
- **Data Flow**: Receives orders from the **Order Service**, executes trades, stores trade details in the database, and sends execution confirmation to the **Client**.

5.7 Market Data Service

- **Functionality:** The **Market Data Service** provides real-time market data (e.g., stock prices, bid/ask spreads) to the platform and users.
- **Responsibilities**:
    - Fetch and process live market data streams from liquidity providers and exchanges.
    - Provide real-time updates to traders about price changes, volumes, and other relevant data.
    - Store historical market data in **Elasticsearch** for querying.
- **Data Flow**: Receives market data from external sources, processes it, and makes it available to **Order Service** and **API Gateway** for display to users.

5.8 Risk Management Service

- **Functionality:** The **Risk Management Service** monitors and enforces risk-related policies, including margin checks, exposure limits, and stop-loss triggers.
- **Responsibilities**:
    - Monitor user positions to ensure they stay within acceptable risk limits.
    - Trigger **Risk Alerts** when margin levels are exceeded or exposure thresholds are breached.
    - Provide real-time risk management tools to users and send alerts via the **Notification Service**.
- **Data Flow**: Accesses user portfolio and order data from **PostgreSQL** and triggers alerts if risk thresholds are breached. It interacts with the **Notification Service** to inform users.

5.9 Compliance Service

- **Functionality:** The **Compliance Service** ensures that all trades and orders adhere to financial regulations, including anti-money laundering (AML) checks and trade reporting.
- **Responsibilities**:
    - Verify trades for regulatory compliance, such as KYC (Know Your Customer) and AML checks.
    - Generate compliance reports, audit trails, and tax reports.
    - Ensure that all orders and trades meet industry standards for financial regulations.
- **Data Flow**: Retrieves transaction data from the **Trade Service**, performs checks, stores compliance records in **PostgreSQL**, and generates reports.

5.10 Notification Service

- **Functionality:** The **Notification Service** sends real-time notifications to users about their orders, trades, risk alerts, and market updates.
- **Responsibilities**:
    - Generate and send notifications (e.g., order status, trade execution, alerts).
    - Store notifications temporarily in **Redis** for fast access.
    - Ensure notifications are sent immediately when certain events (like trade execution) occur.
- **Data Flow**: Receives notification requests from services like **Trade Service**, **Order Service**, and **Risk Management**, stores notifications in **Redis**, and sends them to users via email, push, or in-app notifications.

5.11 Databases Choice

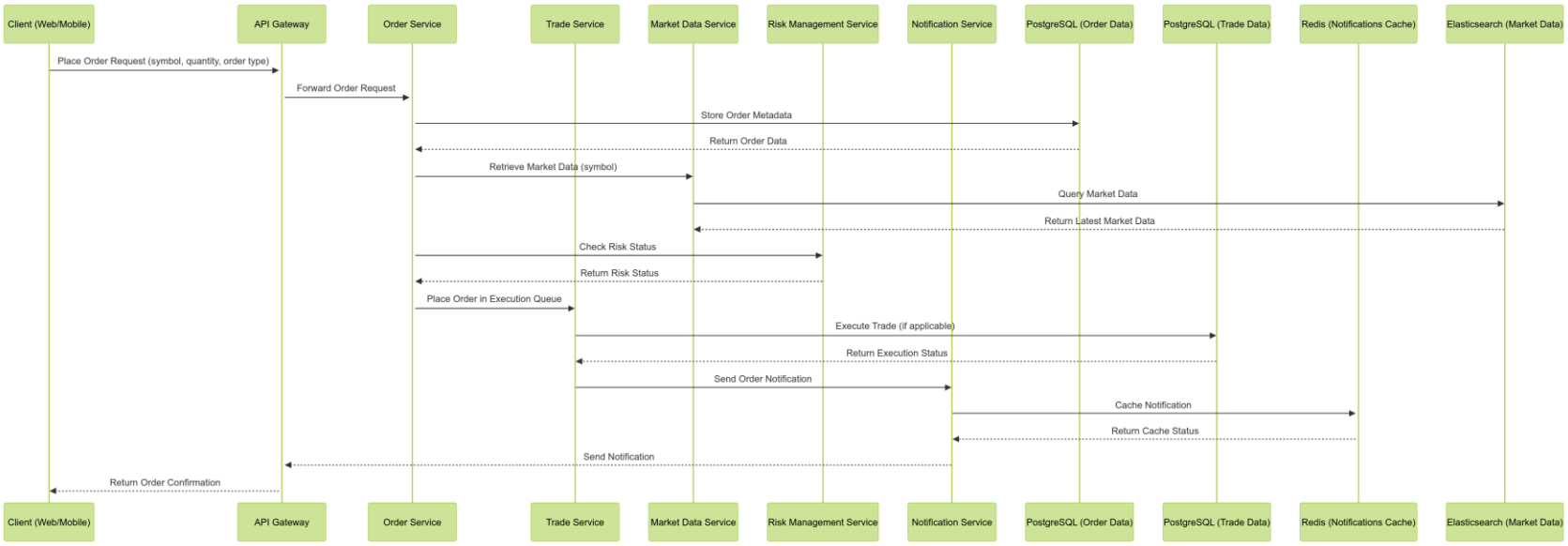
- **PostgreSQL:** Stores structured data such as user profiles, orders, trades, risk data, and compliance records. Provides ACID compliance for transactional integrity.
- **Elasticsearch**: Stores and indexes **market data** for fast search and retrieval, particularly useful for real-time price updates and historical market analysis.
- **Redis**: Used to cache **notifications** and **user session data** for quick access, improving performance and reducing load on the main databases.

6. Request Flows

This section highlights the interaction and information transfer between components and databases by giving example of the flow of execution from key use cases

6.1 Order Placement Flow

- The **Client** places an order through the **API Gateway.**
- The **Order Service** stores the order details in the **PostgreSQL Orders Database**.
- The **Market Data Service** fetches relevant market data (e.g., price, bid/ask) from **Elasticsearch**.
- The **Risk Management Service** performs checks for order execution
- The order is forwarded to the **Trade Service** for execution.
- Once executed, the **Notification Service** sends the order confirmation to the user.



7. Detailed Component Design

This section revisits key components and breaks them down in details

7.1 Order Management System (OMS)

The **Order Management System (OMS)** plays a critical role in the stock trading platform by handling user orders, ensuring efficient execution, and maintaining order lifecycles. Here's a deeper dive into its functionalities:

Core Responsibilities:

- Order Placement: Receives user orders specifying stock symbol, quantity, price (for limit orders), and order type (market, limit, stop-loss, etc.).
- Order Validation: Validates orders against user account information, available funds (for buying), and other pre-defined limits.
- Risk Management Integration: Interacts with the Risk Management component to evaluate potential risks associated with the order before execution.
- Order Routing: Based on order type and market conditions, routes orders to the appropriate connector (exchange or liquidity provider)
- Smart Order Routing (SOR): Implement Algorithms that utilize Order Routing and implement dynamic order splitting that will split the order into the proper exchanges/lps based on liquidity and risk.
- Order Execution: Sends orders to the chosen execution venue for matching with available buy/sell orders.
- Order Status Tracking: Keeps track of the order status (pending, filled, partially filled, cancelled).
- Order Management: Allows users to modify or cancel open orders before execution.
- Order History: Provides users with a record of past orders, including execution details and timestamps.

Edge Case: Handling Simultaneous Orders

Ensuring fair execution priority for simultaneous orders from multiple users is crucial for a robust **OMS**. Here are two common approaches:

1. **Timestamp-Based Order Queuing:** Orders are placed in a queue based on the exact time of receipt (microsecond precision). Orders are executed in the order they appear in the queue, ensuring fairness.

2. **Price-Time Priority:** Orders with better prices (limit orders closer to the National Best Bid and Offer) are prioritized within the timestamp queue. This approach rewards aggressive orders while maintaining a fair execution sequence for orders placed at the same time and price.

The chosen approach depends on the platform's focus (fairness vs. price priority) and regulatory requirements.

Challenges:

- **Partial Execution**: Ensure that partially executed orders remain in the system until fully processed or canceled.
- **Latency**: Handle high-frequency trades with low-latency execution using **asynchronous processing**.

7.2 Market Data Service

The **Market Data Service** is responsible for collecting and providing real-time market data, such as stock prices, volumes, bid/ask spreads, and other market information

Core Responsibilities:

- **Market Data Collection:** Collect, parse and normalize live market data streams from liquidity providers and exchanges
- **Market Data Aggregation:** Aggregate and display market data from multiple sources
- **Historical Data:** Fetch and provide historical market data for analysis

Challenges:

- **Data Feed Failures**: Handle failures from data providers by implementing backup feeds or fallback mechanisms.
- **High Traffic**: Manage high-frequency data streams during volatile market conditions by employing **sharding** and **caching**.

7.3 Trade Execution Engine (TEE)

The Trade Execution Engine (TEE) sits at the heart of order execution within the crypto trading platform. Here's a breakdown of its design considerations, technology choices, and scaling strategies:

Design Principles:

- Low Latency: Every millisecond counts in order execution. The TEE needs to be optimized for low latency communication with exchanges and other execution venues.
- High Availability: The TEE should be highly available to ensure uninterrupted order processing and execution.
- Scalability: The system needs to handle high volumes of orders efficiently and scale to accommodate increasing trading activity.
- Resilience: The TEE should be able to handle network disruptions, exchange outages, and partial order fills gracefully.

8. Trade offs/Tech choices

C++ vs Rust: Both are great but I I chose **C++.** It is crucial to choose a language that offers parallel/asynchronous programming. both languages offer low level control, manual memory management and the ability to call inline assembly. Benchmark tests for performance: they both deliver the best results in terms of execution time, memory efficiency, and low latency.

PostgreSQL vs NoSQL (Elasticsearch): Chose **PostgreSQL** for structured transactional data (orders, trades, users) for strong ACID compliance. Use **Elasticsearch** for fast search and retrieval of market data due to its high scalability and ability to handle real-time queries efficiently.

Horizontal Scaling vs Vertical Scaling: Opted for **horizontal scaling** of services like **Order Service** and **Trade Service** to handle high traffic during market hours, allowing dynamic scaling based on load rather than relying on vertical scaling, which has limited scalability.

Synchronous vs Asynchronous Processing: Used **asynchronous processing** for order execution and trade matching to avoid blocking during peak times, but some critical operations (like order creation) are processed synchronously to ensure data consistency.

PostgreSQL vs Redis for Caching: Used **Redis** to cache notifications and session data to reduce database load and improve latency. However, **PostgreSQL** is used for transactional data, as it guarantees data integrity.

9. Failure Scenarios/Bottlenecks

- **Database Overload:** High order volume can overwhelm **PostgreSQL** leading to slow queries.
  Mitigation: Use **read replicas** and **sharding** to distribute the load.
- **Network Latency:** Market data and trades might experience delays.
  Mitigation: Use **FIX, websockets** and **CDN** for fast, real-time data delivery.
- **Rate Limiting:** During peak trading hours, too many requests can cause system slowdowns.
  Mitigation: Implement **rate limiting** and **backpressure** mechanisms.
- **Concurrency Issues**: Multiple simultaneous orders may result in race conditions.
  Mitigation: Use **optimistic concurrency control** and **locking** mechanisms to avoid conflicts.
- **Cache Invalidation**: Stale data in **Redis** could lead to inaccurate notifications.
  Mitigation: Implement **cache expiration** and **TTL** to ensure up-to-date data.
- **API Gateway Bottlenecks**: Heavy traffic might overwhelm the **API Gateway**.
  Mitigation: Use **auto-scaling** and **load balancing** to handle high traffic volumes.
- **Trade Matching Failures**: Failed or partial trade executions can leave orders in an inconsistent state.
  Mitigation: Ensure **transactional integrity** and use **atomic operations** for matching.
- **Real-Time Data Feed Failure**: Loss of market data can disrupt trading.
  Mitigation: Use **backup data providers** and **retries** to ensure data availability.

10. Future Improvements

- **Improvement:** Implement **machine learning models** for predictive analytics and smarter trade recommendations.
  Mitigation: Use edge computing to process data locally and minimize latency.
- **Improvement**: Introduce **multi-cloud architecture** for better availability and disaster recovery.
  Mitigation: Replicate data across multiple regions to ensure failover during network or server failures.
- **Improvement:** Introduce **High Performance Computing (HPC)** by deploying specialized hardware like GPUs, FPGAs and low latency network interfaces.
- **Improvement**: Enhance **real-time data pipelines** for faster processing and fewer bottlenecks.
  Mitigation: Implement **Kafka** or **AWS Kinesis** for high-throughput streaming data handling.
- **Improvement**: Integrate **blockchain** for immutable audit trails in trading and compliance.
  Mitigation: Ensure **multi-layer security** with encryption and continuous monitoring to avoid data tampering.
- **Improvement**: Use **AI-powered monitoring** tools to detect and alert on anomalies in real-time.
  Mitigation: Implement **auto-healing** mechanisms to resolve issues automatically before they impact performance.

Appendix A: Record of Changes

Table 1 - Record of Changes

Version Number	Date	Author/Owner	Description of Change
1.0	*18/03/2025*	*Joe Archondis*	*1st Draft*

Appendix B: Acronyms

Table 2 - Acronyms

Acronym	Literal Translation
API	*Application Programming Interface*
HFT	*High Frequency Trading*
JWT	*Json Web Token*
LP	*Liquidity Provider*
OMS	*Order Management System*
SOR	*Smart Order Router*
TEE	*Trade Execution Engine*

Appendix C: Glossary

Instructions: Provide clear and concise definitions for terms used in this document that may be unfamiliar to readers of the document. Terms are to be listed in alphabetical order.

Table 3 - Glossary

Term	Acronym	Definition
<Term>	*<Acronym>*	*<Definition>*
<Term>	*<Acronym>*	*<Definition>*
<Term>	*<Acronym>*	*<Definition>*

Appendix D: Referenced Documents

Table 4 - Referenced Documents

Document Name	Document Location and/or URL	Issuance Date
Benchmark for c++ and rust	https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html	*18/03/2025*
Diagram Editor	*https://mermaid.live*	*15/03/2025*