**Trading Strategy System Requirements Documentation**

This document outlines the requirements for a conversational multi-agent AI trading strategy system designed to allow users to define personalized trading strategies through an interactive conversational agent. The system leverages modern technologies to ensure performance, scalability, and security, with a Neo4j graph database at its core to manage relationships between components, supporting validation, intelligent suggestions, and backtesting.

**1. Introduction**

The trading strategy system enables users to create custom trading strategies via a conversational agent that collects inputs, validates them, and ensures the strategy is complete and correctly specified. The system is designed to be modular, secure, and scalable, with real-time data handling and robust backtesting capabilities. This document provides a detailed blueprint for developers and a reference for stakeholders, covering the system’s architecture, functionality, and implementation details.

**2. Technology Stack**

The system is built using a modern, scalable technology stack to ensure high performance, reliability, and ease of development:

* **Frontend**: React (JavaScript) – Delivers a dynamic and responsive user interface.
* **Backend**: FastAPI (Python) – Provides high-performance processing, API management, and asynchronous operations.
* **Agentic Layer**: LangChain with Claude 3.7 Sonnet (LLM) – Drives the conversational AI and agent coordination.
* **Data Layer**:
  + **Neo4j (Graph Database)** – Manages structured data and relationships efficiently.
  + **InfluxDB (Time-Series Database)** – Stores and manages time-series data like OHLCV (Open, High, Low, Close, Volume).
* **Additional Technologies**:
  + **RESTful APIs** – Facilitate communication between frontend and backend.
  + **WebSockets** – Enable real-time data streaming.
  + **Python’s multiprocessing** – Optimizes parallel processing for feature calculations.

**3. Key Components**

The system comprises core components represented as nodes in the Neo4j graph database. Relationships between components are illustrated below:

mermaid

CollapseWrapCopy

graph TD

StrategyType -->|COMMONLY\_USES| Indicator

StrategyType -->|SUITABLE\_FOR| Instrument

StrategyType -->|TYPICAL\_FREQUENCY| Frequency

StrategyType -->|RECOMMENDED\_BACKTESTING| BacktestingMethod

Instrument -->|APPROVED\_FOR| Timeframe

Indicator -->|HAS\_PARAMETER| Parameter

Indicator -->|USED\_IN| Condition

BacktestingMethod -->|HAS\_PARAMETER| Parameter

**Component Definitions**

* **StrategyType**: Defines the trading approach.
  + *Properties*:
    - name: String (e.g., "momentum")
    - description: String (e.g., "Capitalizes on price momentum")
  + *Relationships*:
    - -[:COMMONLY\_USES]-> Indicator (e.g., momentum uses RSI)
    - -[:SUITABLE\_FOR]-> Instrument (e.g., high-volatility stocks)
    - -[:TYPICAL\_FREQUENCY]-> Frequency (e.g., daily)
    - -[:RECOMMENDED\_BACKTESTING]-> BacktestingMethod (e.g., walk-forward)
* **Instrument**: The asset to trade.
  + *Properties*:
    - symbol: String (e.g., "BTCUSDT")
    - type: String (e.g., "crypto")
    - data\_source: String (e.g., "Binance API")
  + *Relationships*:
    - -[:APPROVED\_FOR {status: String, data\_source: String}]-> Timeframe (e.g., approved for daily via Binance)
* **Frequency**: Time interval for trading decisions.
  + *Properties*:
    - name: String (e.g., "daily")
* **Indicator**: Technical analysis tool.
  + *Properties*:
    - name: String (e.g., "RSI")
    - description: String (e.g., "Measures momentum")
  + *Relationships*:
    - -[:HAS\_PARAMETER]-> Parameter (e.g., RSI has period)
    - -[:USED\_IN]-> Condition (e.g., RSI in entry condition)
* **Parameter**: Configurable setting.
  + *Properties*:
    - name: String (e.g., "period")
    - default\_value: Number/String (e.g., 14)
    - min\_value: Number (e.g., 5)
    - max\_value: Number (e.g., 50)
* **Condition**: Trade entry/exit rule.
  + *Properties*:
    - logic: String (e.g., "RSI > 70")
    - type: String ("entry" or "exit")
* **PositionSizing**: Determines trade size.
  + *Properties*:
    - method: String (e.g., "percent")
    - value: Number (e.g., 2)
* **RiskManagement**: Limits losses or secures profits.
  + *Properties*:
    - stop\_loss: Number (e.g., 5)
    - take\_profit: Number (e.g., 10)
    - max\_positions: Number (e.g., 3)
* **BacktestingMethod**: Evaluates strategy performance.
  + *Properties*:
    - type: String (e.g., "walk-forward")
  + *Relationships*:
    - -[:HAS\_PARAMETER]-> Parameter (e.g., in-sample period)

**4. User Interaction Flow**

The conversational agent guides users through strategy creation using a structured process. The flowchart below illustrates this flow:

mermaid

CollapseWrapCopy

flowchart TD

A[Start] --> B[Select Strategy Type]

B --> C[Select Instrument]

C --> D[Select Frequency]

D --> E[Select Indicators and Parameters]

E --> F[Define Entry and Exit Conditions]

F --> G[Set Position Sizing and Risk Management]

G --> H[Choose Backtesting Method]

H --> I[Validate and Confirm Strategy]

I --> J[End]

**Interaction Steps**

1. **Select Strategy Type**
   * *Prompt*: "What type of strategy? (e.g., momentum, mean reversion)"
   * *Input*: strategy\_type: String
   * *Validation*: Check if the strategy type exists in the database.
2. **Select Instrument**
   * *Prompt*: "Which instrument? Momentum suits volatile assets like TSLA."
   * *Input*: symbol: String
   * *Validation*: Confirm the instrument is approved for the selected timeframe.
3. **Select Frequency**
   * *Prompt*: "What frequency? Momentum typically uses daily or hourly."
   * *Input*: frequency: String
   * *Validation*: Ensure frequency aligns with strategy type recommendations.
4. **Select Indicators and Parameters**
   * *Prompt*: "Which indicators? Momentum often uses RSI (period 14)."
   * *Input*: indicators: [{name: String, parameters: [{name: String, value: Number}]}]
   * *Validation*: Verify parameter values are within acceptable ranges.
5. **Define Entry and Exit Conditions**
   * *Prompt*: "Entry condition? (e.g., 'RSI > 70')"
   * *Input*: conditions: [{logic: String, type: String}]
   * *Validation*: Ensure at least one entry and one exit condition are provided.
6. **Set Position Sizing and Risk Management**
   * *Prompt*: "Position size? (e.g., 2% of portfolio)"
   * *Input*: position\_sizing: {method: String, value: Number}, risk\_management: {stop\_loss: Number, take\_profit: Number, max\_positions: Number}
   * *Validation*: Confirm all fields are specified.
7. **Choose Backtesting Method**
   * *Prompt*: "Backtesting method? Momentum recommends walk-forward."
   * *Input*: backtesting\_method: String
   * *Validation*: Ensure the method is compatible with the strategy type.

**5. User Accounts and Authentication**

The system includes user accounts and authentication to secure access and personalize experiences.

**5.1. Authentication**

* **Technology**: FastAPI’s OAuth2 support with JWT tokens.
* **Functionality**:
  + Users can register, log in, and log out.
  + JWT tokens manage sessions and authenticate API requests.
  + Passwords are hashed and stored securely.

**5.2. User Data Storage**

* **Database**: SQLite (MVP), with plans to upgrade to PostgreSQL for scalability.
* **Data Stored**: User profiles, strategy templates, and backtesting results.

**6. Real-Time Data Handling**

The system streams real-time market data to the frontend using WebSockets.

**6.1. WebSocket Implementation**

* **Technology**: FastAPI’s native WebSocket support.
* **Functionality**:
  + Streams real-time data (e.g., price updates) to the frontend.
  + Supports multiple concurrent connections.
* **Use Case**: Live price charts or real-time backtest updates.

**7. Security and Compliance**

Security is critical for the trading platform, ensuring data protection and regulatory compliance.

**7.1. Security Measures**

* **Authentication**: OAuth2 with JWT tokens.
* **Data Encryption**:
  + TLS for data in transit.
  + AES for data at rest.
* **Input Validation**: Prevents injection attacks (e.g., SQL injection, XSS).
* **Rate Limiting**: Mitigates DDoS attacks and API abuse.

**7.2. Compliance**

* **Data Privacy**: Adheres to GDPR and other relevant regulations.
* **Audit Logging**: Tracks user actions and system events for compliance and debugging.

**8. Scalability and Performance**

The system is designed to handle growth and maintain performance under load.

**8.1. Scalability Strategies**

* **Load Balancing**: Distributes traffic across multiple servers.
* **Caching**: Uses Redis for frequently accessed data.
* **Horizontal Scaling**: Adds servers as needed.

**8.2. Performance Optimization**

* **Database Indexing**: Speeds up query performance.
* **Asynchronous Processing**: Leverages FastAPI’s async capabilities for I/O tasks.
* **Efficient Data Handling**: Minimizes data transfer and processing overhead.

**9. Validation Rules**

* **Compatibility**: Ensures indicators align with the strategy type (e.g., RSI for momentum).
* **Completeness**: Verifies all required inputs (strategy type, instrument, frequency, conditions) are provided.
* **Parameter Ranges**: Checks that parameter values are within defined limits.
* **Instrument Approval**: Confirms the instrument is approved for the selected frequency and data source.

**10. Graph Database Structure**

* **Nodes**: StrategyType, Instrument, Frequency, Indicator, Parameter, Condition, PositionSizing, RiskManagement, BacktestingMethod, Timeframe
* **Relationships**: As depicted in the Mermaid diagram in Section 3.
* **Example Node**:

json

CollapseWrapCopy

{

"Indicator": {

"name": "RSI",

"description": "Relative Strength Index",

"parameters": [

{"name": "period", "default\_value": 14, "min\_value": 5, "max\_value": 50}

]

}

}

**11. Scalability and Flexibility**

* **Modular Design**: Allows adding new strategy types, indicators, or backtesting methods as standalone nodes.
* **Generic Relationships**: Uses flexible connections (e.g., HAS\_PARAMETER) to support various components.
* **Versioning**: Tracks changes with version properties on nodes.

**12. Development Guidance**

* **Phase 1**: Implement core components (StrategyType, Instrument, Frequency, Indicator, Parameter, Condition, basic backtesting).
* **Phase 2**: Add advanced features (custom conditions, multi-database support, enhanced backtesting).
* **Key Tasks**:
  + Generate Neo4j schema from node/relationship definitions.
  + Build agent logic using interaction flow and validation rules.
  + Develop validation queries (e.g., instrument approval).

**13. Backtesting Requirements**

Backtesting evaluates strategy performance using historical data, supporting multiple methods and efficient computation.

**13.1. Data Partitioning**

* **Simple Backtesting**: Uses a single historical window with fixed parameters.
* **Walk-Forward Optimization**: Divides data into in-sample (IS) and out-of-sample (OOS) windows (anchored or rolling).
* **Cross-Validation**: Splits data into multiple folds for training and testing.

**13.2. Feature Calculation and Optimization**

* **Feature Calculation**: Computes indicators (e.g., RSI with period 14) based on parameters.
  + Includes lead-in periods (e.g., 14 days of prior data for RSI).
* **Parallel Processing**: Uses Python’s multiprocessing for parallel feature matrix calculations.
* **Memory Efficiency**: Loads historical data once and shares it across processes.
* **IS Optimization**: Computes feature matrices for all parameter combinations in parallel and selects the best.
* **OOS Evaluation**: Applies optimized parameters to evaluate OOS performance.

**13.3. Example Workflow (Walk-Forward Optimization)**

1. Load historical data for the current IS window.
2. Define a parameter search space (e.g., RSI period from 10 to 20).
3. Compute feature matrices in parallel for each parameter set (e.g., RSI\_10, RSI\_11, ..., RSI\_20).
4. Evaluate strategy performance to identify the best parameter set.
5. Apply the best parameters to compute and evaluate the OOS feature matrix.

**14. Hybrid Multi-Agent Architecture**

The system uses a hybrid multi-agent architecture with two layers: the **data layer** and the **agentic layer**.

**14.1. Data Layer**

* **Technology**: Neo4j graph database.
* **Responsibilities**:
  + Stores structured information (strategy types, instruments, indicators, etc.).
  + Supports validation and suggestions.
  + Persists strategy templates and backtesting results.

**14.2. Agentic Layer**

* **Technology**: LangChain with Claude 3.7 Sonnet.
* **Responsibilities**:
  + **Conversational Agent**: Collects and interprets user inputs.
  + **Validation Agents**: Ensure input correctness and completeness.
  + **Data and Feature Agents**: Retrieve data and compute indicators.
  + **Code Agents**: Generate and execute strategy code.
  + **Feedback Agent**: Evaluates performance and suggests improvements.

**14.3. Interaction Between Layers**

* **Agentic to Data**: Queries or updates the data layer.
* **Data to Agentic**: Supplies structured data for processing.

**14.4. Benefits**

* **Clarity**: Separates data management from logic.
* **Scalability**: Enables independent layer upgrades.
* **Efficiency**: Combines structured data with automated execution.