**Started requirement understanding and concepts:**

**Started preparing document**

**Started preparing system architecture design**

**Started preparing drawio diagram for system architecture**

**Started preparing swimlane flow diagram**

**Started preparing wireframe**

**Started backend development**

**Project requirement understanding:**

1. I have to use provided Binance WebSocket streams emitting tick data
2. Ingest raw tick data from webscoket and store it . Sampling should be based on selected timeframe
3. Based on sampled data we have to process/analyse that data
   1. Hedge Ratio (via Ordinary Least Squares Regression)
   2. The resulting Spread time series
   3. The Z-Score of the Spread
   4. Stationarity check via Augmented Dickey-Fuller (ADF) Test
   5. Rolling Correlation over a configurable window
4. Create frontend interactive dashboard which will consume the analytics data and show to user in real time
5. we have to allow alert functionality and download option and data upload option

Technical & Design:

1. The backend must be written in Python.
2. The system design must be loosely coupled and extensible, allowing for future additions of new analytics or data sources with minimal rework.
3. The application must be runnable via a single local command and be delivered with clear documentation (README.md) and an architecture diagram.

Based on above understanding I started the development

**My understanding for definitions of Concepts used in project:**

Before starting the project I have to understand the trading concepts which are going to be used in the project. Below is my understanding about the concepts.

1. Tick

A tick represents one transaction on Binance. It tells us at what time in the world a certain amount of an asset was bought at what price.

1. Hedge Ratio

The Hedge Ratio is calculated between two related assets.

It answers: "How much of one asset do I need to buy to balance 1 unit of another asset to create a neutral position?"

1. The Spread

The hedge ratio we calculated is used here.

Formula: Spread = Price of Asset A - (Hedge Ratio \* Price of Asset B)

Why we need spread: If we plot two assets, let's say Pepsi and Coke, then we get two separate lines. They are related, so if a factor like the price of sugar rises, then they are both going to rise. But there are other factors which can affect a single asset. As they are related, they are eventually going to come back to their normal price relationship. These sudden fluctuations are what an investor sees as an opportunity to make a profit.

To trade this, we cannot just rely on simple subtraction because a 5-unit difference is not the same when prices are high vs. when they are low. This is not reliable.

Here, the Spread is used, which uses the hedge ratio to create one single line.

Now the scenario is: when the price of one asset goes up, it causes the Spread to vary. Let's say the price of Coke is overpriced; then what the investor does is sell the overpriced Coke and buy Pepsi, because they expect the Spread is going to be 0 again. So when the Spread reaches 0, the investor can close the position and make a profit.

1. Z-Score

What it is: A number that tells us how many standard deviations the Spread is away from its average.

Purpose: It's our main signal. A high Z-score (e.g., > 2) signals that the Spread is unusually high and might be a good time to sell. A low Z-score (e.g., < -2) signals it's unusually low and might be a good time to buy.

1. ADF (Augmented Dickey-Fuller) Test: (This is a check before you start trading)

In a spread strategy, we trade because we expect the spread between two assets is going to come to zero.

The ADF test is performed before trading to check whether that spread is likely going to come to 0 or if it is just going to increase continuously.

If the p-value is LOW (less than 0.05), then the spread is "stationary" (it is likely going to return to its average).

If the p-value is HIGH (more than 0.05), it is "non-stationary" (it is not likely to return).

1. Rolling Correlation: (This is a check to use while the trade is going on)

Correlation is a number from -1 to +1 that measures how much two things move together. +1 means they move in perfect lockstep. 0 means there's no connection.

"Rolling" means we calculate this based on a recent window, like the last 60 minutes.

If the correlation is high (for example, above 0.7), it confirms the relationship between the assets is strong and our strategy is still valid.

If the correlation is low (like 0.2), this is a warning that the relationship has weakened, and we should be careful or think about exiting the trade.

**System architecture and design and trade-offs:**

We have four main components in the architecture:

1. Data Ingestor: An independent background process responsible solely for data collection.
2. Database
3. Backend API Server
4. Frontend Client

In above as we can see we are using Client-Server Architecture for separation of concerns, principles of loose coupling and scalability.

|  |  |
| --- | --- |
| Component | Technology |
| Backend Framework | FastAPI |
| Frontend Framework | React |
| Charting Library | TradingView Lightweight Charts |
| Database | SQLite |

Reason for selection of technology for component and trade-offs:

1. Backend: FastAPI

Why selected FastAPI instead of a framework like Django:

1. FastAPI is used for high-performance APIs with simple, less complex code. This allows for quick development and debugging while still achieving the required high performance.
2. A framework like Django is not as suitable for this specific case. It requires more boilerplate code, and more code can mean more chances for error. Django is a full-stack tool, and we only need the API part.
3. Our project involves using WebSockets for live updates. FastAPI is built on ASGI (Asynchronous Server Gateway Interface), which is designed to handle many connections and long-running tasks like a WebSocket asynchronously, without getting blocked.
4. Django, on the other hand, was built on WSGI (Web Server Gateway Interface), which is synchronous. To handle a WebSocket in Django, we would need an extra library like Django Channels, adding more complexity.
5. FastAPI's Dependency Injection system allows us to write cleaner and more modular code. For example, I can define the database connection I need in a function's signature, and FastAPI will manage it. This makes it very easy to swap things later, like using a different database for testing.
6. FastAPI also gives us automatic data validation and API documentation for free. By defining the data models with standard Python types, it validates all incoming requests and generates an interactive API doc page. This saves a lot of time and makes the API more robust.

2. Database: SQLite

Why selected SQLite:

1. We selected it because it is a serverless database that runs as a local file on the same machine. This is the most important reason because it allows us to meet the "single-command run" requirement. It requires no extra setup or installation from the user, unlike Docker or a separate PostgreSQL server.
2. The performance is more than enough for this project. Since it's a local file, read/write operations are extremely fast. For the small queries we need to run like getting the last 100 ticks, SQLite with a proper index is incredibly efficient and will not be the bottleneck in our 500ms latency budget.

3. Frontend: React with TradingView Lightweight Charts

1. Using a separate React app creates strong loose coupling. The frontend is its own project. If we have a scenario where we need to add a new chart, we just have to create a new component in the React app and have it use an existing or new API endpoint. We don't need to update the entire application, just the frontend code.
2. This loose coupling works both ways. If we have a scenario where we have to change the backend technology (e.g., from Python to Go), we can easily do it. As long as the new backend provides the same API endpoints, we can just "plug in" the new service and the React frontend will work without any changes.
3. We chose the TradingView Lightweight Charts library specifically because it is a professional, high-performance tool built for financial data. It already has all the components required to show candlestick charts with high-speed interactivity (zooming and panning). This follows the "clarity over complexity" principle by not reinventing a complex component from scratch.