This is an exciting and ambitious project! Given the complexity and your goals, here's a high-level design approach that will allow you to start small and scale over time, while maintaining the modularity and decoupling you desire.

### 1. \*\*System Architecture Overview\*\*

#### 1.1 Core Components:

- \*\*Data Source Layer\*\*: Interfaces with various external data providers (e.g., Binance, other exchanges, stock APIs).

- \*\*Data Processing Layer\*\*: Handles the collection, transformation, and storage of data in the desired format.

- \*\*Database Layer\*\*: Abstracts the underlying database technology, providing an interface to store and retrieve data.

- \*\*Analysis Engine\*\*: Computes entry and exit signals based on different strategies and time frames.

- \*\*Time Frame Aggregation\*\*: Aggregates minute-level data into higher time frames (e.g., 5m, 15m, 1H).

- \*\*Backtesting Module\*\*: Runs simulations based on historical data and various time frames.

- \*\*Scheduler/Orchestrator\*\*: Manages the execution of tasks, such as data collection and analysis, possibly using AWS Lambda or Step Functions.

- \*\*API Layer\*\*: Provides a way to interact with the system, retrieve results, and manage configurations.

### 2. \*\*Data Source Layer\*\*

#### 2.1 Modular Interface:

- \*\*DataSource Interface\*\*: Define an interface that all data sources (e.g., Binance, other exchanges) must implement.

- \*\*Concrete Implementations\*\*: Implement the interface for each data source, allowing you to add or switch data sources without affecting other parts of the system.

```python

class DataSourceInterface:

def fetch\_minute\_data(self, symbol, start\_time, end\_time):

raise NotImplementedError

class BinanceDataSource(DataSourceInterface):

def fetch\_minute\_data(self, symbol, start\_time, end\_time):

# Implementation for fetching data from Binance

```

### 3. \*\*Database Layer\*\*

#### 3.1 Database Abstraction:

- \*\*Database Interface\*\*: Define an interface that abstracts the interaction with the database (e.g., storing and retrieving price data).

- \*\*Concrete Implementations\*\*: Implement this interface for different database systems (e.g., PostgreSQL, DynamoDB).

```python

class DatabaseInterface:

def store\_price\_data(self, data):

raise NotImplementedError

def retrieve\_price\_data(self, symbol, time\_frame, start\_time, end\_time):

raise NotImplementedError

class PostgreSQLDatabase(DatabaseInterface):

def store\_price\_data(self, data):

# Implementation for PostgreSQL

```

### 4. \*\*Data Processing Layer\*\*

#### 4.1 Data Transformation:

- \*\*Transformations\*\*: Apply any necessary transformations (e.g., converting time zones, handling missing data) before storing the data.

### 5. \*\*Analysis Engine\*\*

#### 5.1 Strategy Interface:

- \*\*Strategy Interface\*\*: Define an interface for trading strategies that the analysis engine can run.

- \*\*Concrete Strategies\*\*: Implement different strategies as concrete classes.

```python

class StrategyInterface:

def analyze(self, price\_data):

raise NotImplementedError

class SimpleMovingAverageStrategy(StrategyInterface):

def analyze(self, price\_data):

# Implementation of a simple moving average strategy

```

### 6. \*\*Time Frame Aggregation\*\*

#### 6.1 Aggregation Logic:

- \*\*Aggregation Module\*\*: Create a module that can aggregate minute-level data into higher time frames (e.g., 5m, 15m, 1H).

- \*\*Flexible Aggregation\*\*: Allow this module to be easily configured to create different time frames based on user requirements.

### 7. \*\*Backtesting Module\*\*

#### 7.1 Historical Simulation:

- \*\*Backtester Interface\*\*: Define an interface for running backtests on historical data.

- \*\*Modular Backtesting\*\*: Implement various backtesting methodologies that can be plugged into the system.

### 8. \*\*Scheduler/Orchestrator\*\*

#### 8.1 AWS Integration:

- \*\*AWS Lambda\*\*: Use Lambda functions to run tasks such as data collection, analysis, and backtesting.

- \*\*Step Functions\*\*: Use Step Functions to orchestrate complex workflows, such as running multiple analysis strategies in parallel.

### 9. \*\*API Layer\*\*

#### 9.1 Interaction Interface:

- \*\*REST API\*\*: Provide a REST API that allows users to interact with the system, such as retrieving data, running backtests, and fetching analysis results.

- \*\*Configuration Management\*\*: Allow configuration changes (e.g., data sources, strategies) via the API.

### 10. \*\*Security Considerations\*\*

#### 10.1 Authentication and Authorization:

- \*\*Secure API Access\*\*: Implement authentication and authorization mechanisms to ensure that only authorized users can interact with the system.

- \*\*Data Encryption\*\*: Ensure that sensitive data is encrypted both in transit and at rest.

### 11. \*\*Scalability and Performance\*\*

#### 11.1 Horizontal Scaling:

- \*\*Load Balancing\*\*: Use load balancing to distribute the load across multiple instances of the system.

- \*\*Auto-Scaling\*\*: Implement auto-scaling to handle increases in load, especially during peak trading hours.

### 12. \*\*Logging and Monitoring\*\*

#### 12.1 Centralized Logging:

- \*\*Log Aggregation\*\*: Use a centralized logging system to aggregate logs from different components of the system.

- \*\*Real-Time Monitoring\*\*: Implement real-time monitoring and alerting to detect issues early.

### 13. \*\*Deployment Strategy\*\*

#### 13.1 Continuous Integration/Continuous Deployment (CI/CD):

- \*\*CI/CD Pipeline\*\*: Set up a CI/CD pipeline to automate testing, deployment, and scaling.

- \*\*Infrastructure as Code\*\*: Use tools like Terraform or CloudFormation to manage AWS resources as code.

### 14. \*\*Initial Implementation Roadmap\*\*

1. \*\*Phase 1: Core Infrastructure\*\*

- Set up the basic data source and database layers.

- Implement the first data source (e.g., Binance).

- Implement the first database interface (e.g., PostgreSQL).

2. \*\*Phase 2: Data Collection\*\*

- Implement minute-level data collection.

- Add basic time frame aggregation.

- Implement the data storage mechanism.

3. \*\*Phase 3: Analysis Engine\*\*

- Implement the first trading strategy.

- Integrate the strategy with the data collection system.

4. \*\*Phase 4: Backtesting and Expansion\*\*

- Implement the backtesting module.

- Add support for additional data sources and security categories.

5. \*\*Phase 5: AWS Deployment\*\*

- Deploy the system to AWS using Lambda and other cloud-native technologies.

- Implement the API layer for interaction and configuration management.

This approach will allow you to start with a small, modular system and scale it over time. The decoupling of components ensures that you can adapt to changes in technology and requirements with minimal disruption.

Let me know if you need more detailed design or implementation guidance on any specific part of this system!