Problem Statement and Requirements:

**Business Requirements**

**a. Clearly define the problem the system aims to solve.**

The primary objective of this system is to address the need for high-frequency trading by providing robust platform to train , host and execute trainable market data algorithms for high frequency trading.

**b. Specify the functionalities the system needs to provide.**

1. **Order Management**: Allow users to place, modify and cancel orders quickly.
2. **Market Data Integration**: Provide access to market data feeds.
3. **Trade Execution**: Execute trades with minimal delay.
4. **Portfolio Management**: Allow users to manage and monitor their portfolios.
5. **Risk Management**: Assess and mitigate trading risks before, during and after executing trades.
6. **Analytics and Reporting**: Generate analytics and reports on market and the trader's activities.
7. **User Management**: Support user authentication, authorization and profile management.

**c. Identify the target users and their needs.**

1. **Individual Traders**: Need a platform to execute trades at high-frequency and efficiently, with access to data and analytics.
2. **Institutional Traders**: Require a robust and scalable system to handle high volumes of trades, with advanced risk management and reporting capabilities.
3. **Compliance Officers**: Need tools to monitor trading activities and ensure compliance with regulations.

**d. Outline any business goals the system should support.**

1. **Increased Trade Volume**: Enable a higher volume of trades by minimizing latency.
2. **Improved Decision Making**: Provide support for trainable analytics to support better trading decisions.
3. **Risk Mitigation**: Implement effective risk management to minimize potential losses.
4. **User Satisfaction**: Ensure reliable performance to maintain high user satisfaction and retention.

**Non-Functional Requirements**

**a. Define performance requirements like scalability, response time and throughput.**

1. **Scalability**: The system must handle increasing numbers of users and trades without performance degradation.
2. **Response Time**: Trade execution and data retrieval should be performed within milliseconds.
3. **Throughput**: The system should support a high volume of concurrent trade orders and market data updates.

**b. Specify security requirements like authentication, authorization and data encryption.**

1. **Authentication:** Provide for scalable user authentication.
2. **Authorization:** Implement role-based access control.
3. **Data Encryption:** Ensure data protection, encrypting data both at rest and in transit.

**c. Outline maintainability requirements like code modularity, documentation and testing strategies.**

1. **Database Normalization:** Optimize database structure for efficiency and scalability.
2. **Modular Code:** Ensure clear separation of concerns within the system.
3. **Non-Overlapping Functions:** Design with distinct, non-redundant components.
4. **Rigorous Testing:** Implement continuous testing for early detection of issues.
5. **Thorough Documentation:** Provide comprehensive guides for system understanding.

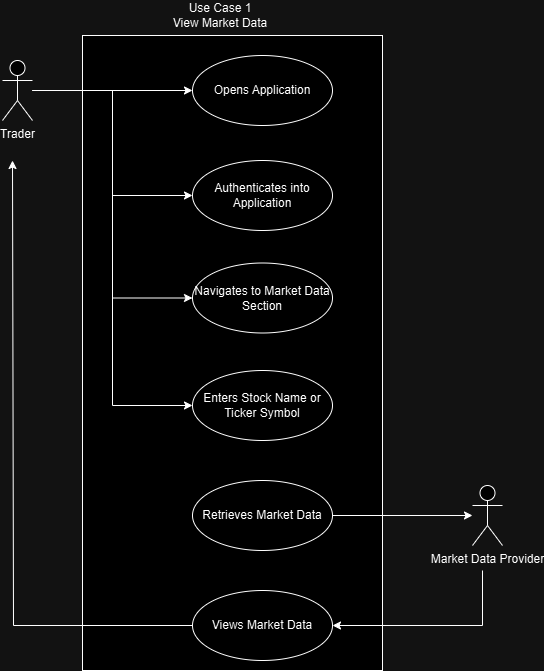
**d. Indicate any other non-functional requirements relevant to the system's success.**

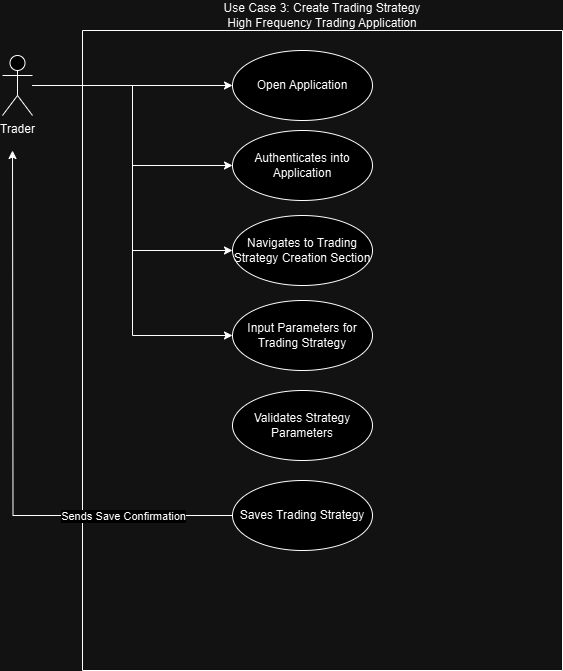
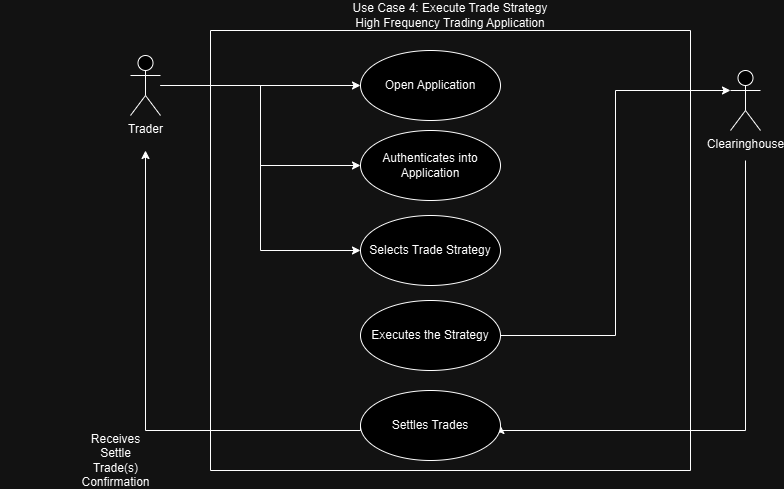
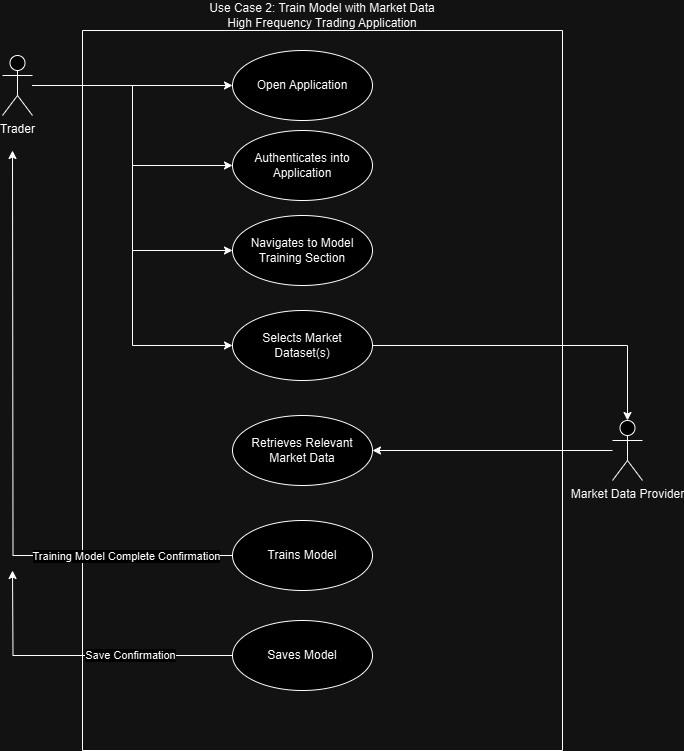
1. **Compliance:** Leverage technologies for adherence to financial regulations.
2. **High Availability:** Ensure redundancy and failover.
3. **Reliability:** Utilize auto scaling and load balancing for consistent performance.
4. **Usability:** Optimize user experience via services tailored to reduce latency.

System Design using Domain Modeling (100 points)

1. UML Use Case Diagram (10 points):

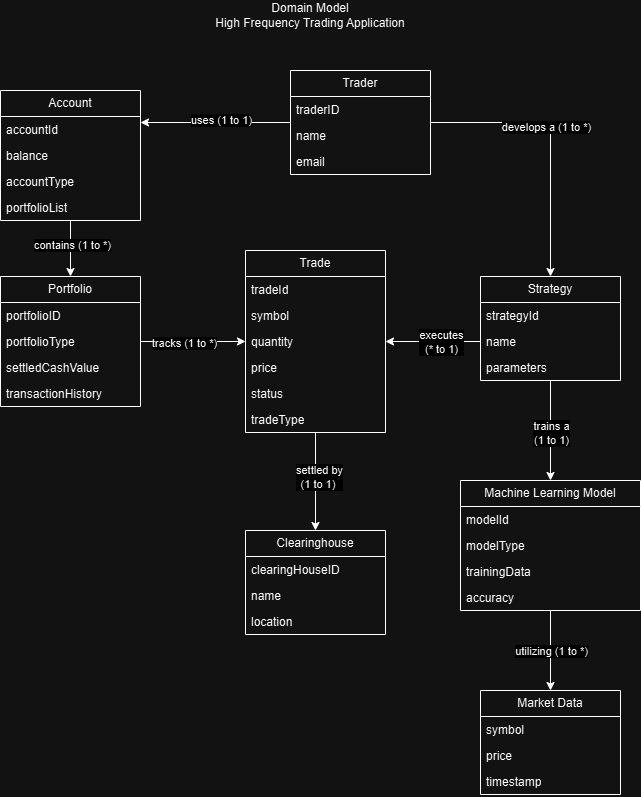
a. Create a visual representation of the system's actors (users and external systems) and their interactions with the system.



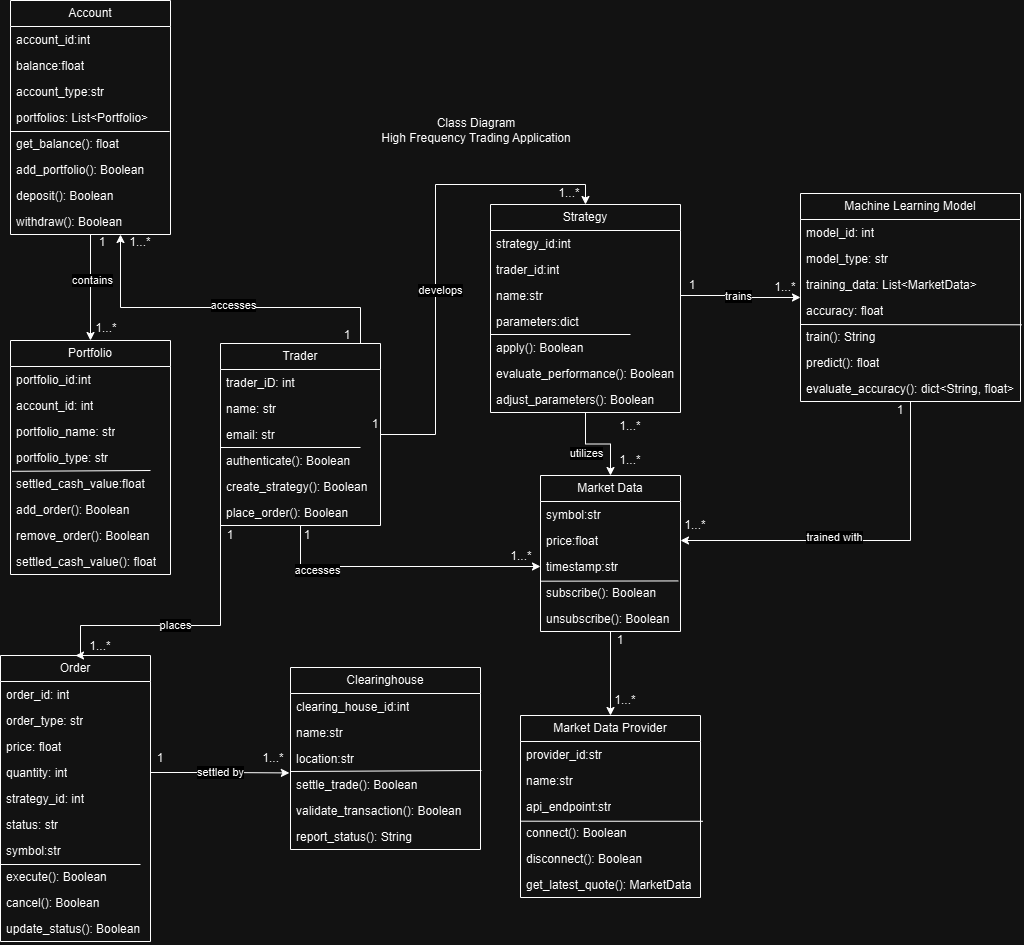


2. UML Domain Model (10 points):

a. Identify key entities and their relationships within the problem domain, independent of any specific technology.

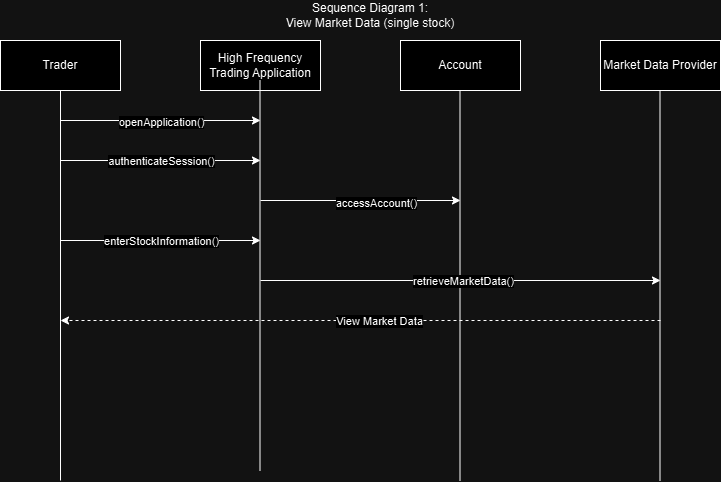


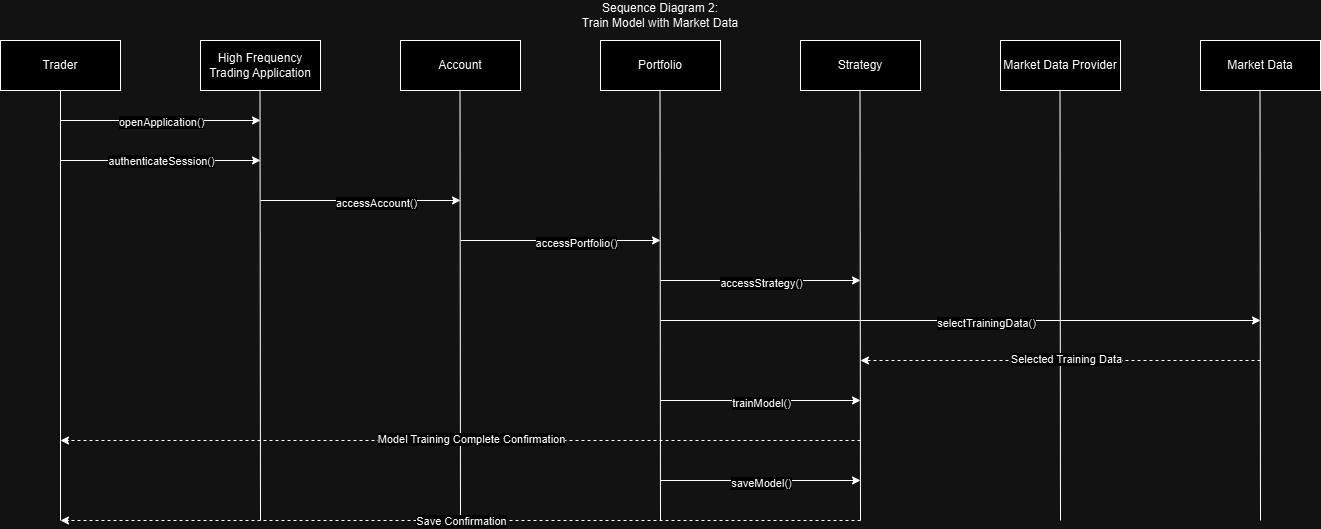
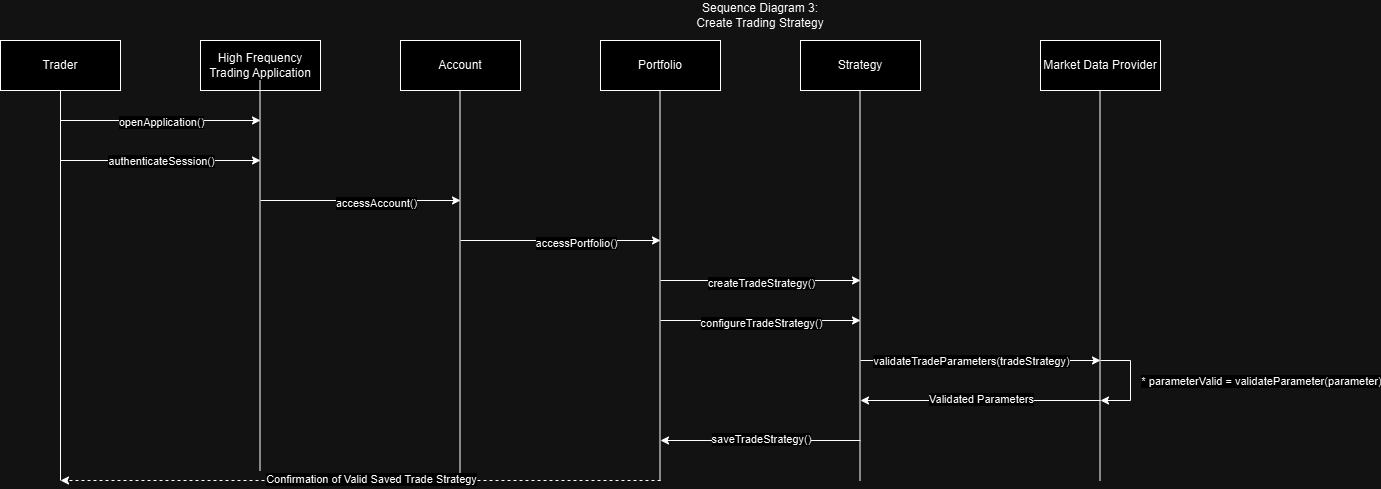
3. UML Class Diagram (10 points):

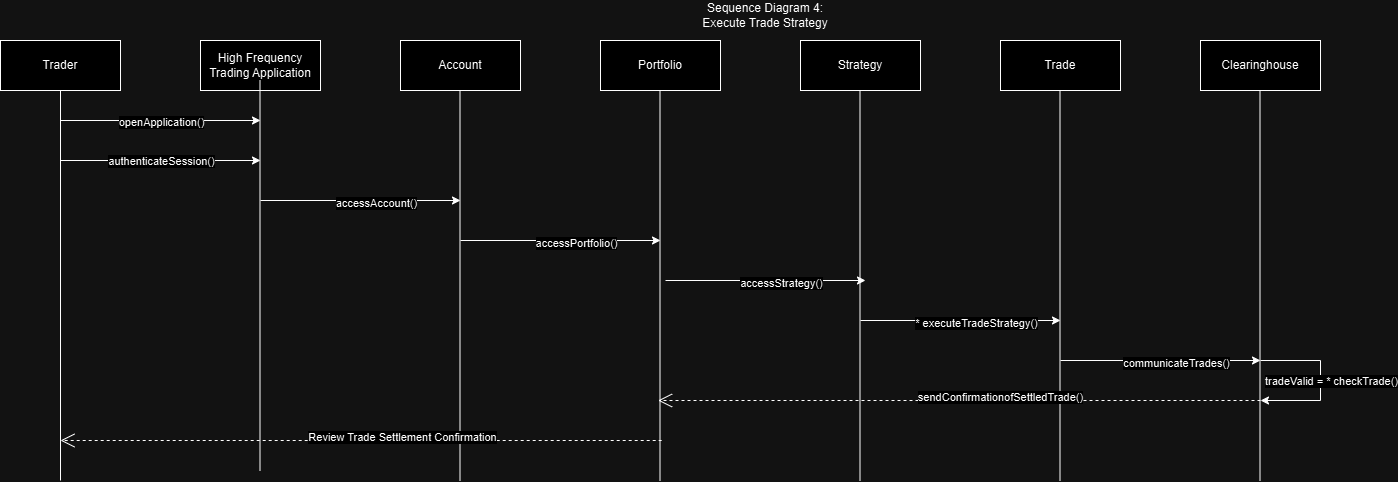
a. Translate the domain model into a set of classes, their attributes and relationships, reflecting the system's functionality.

4. UML Sequence Diagrams (10 points):

a. Show the message flow between objects participating in specific use cases, depicting the interaction sequence.

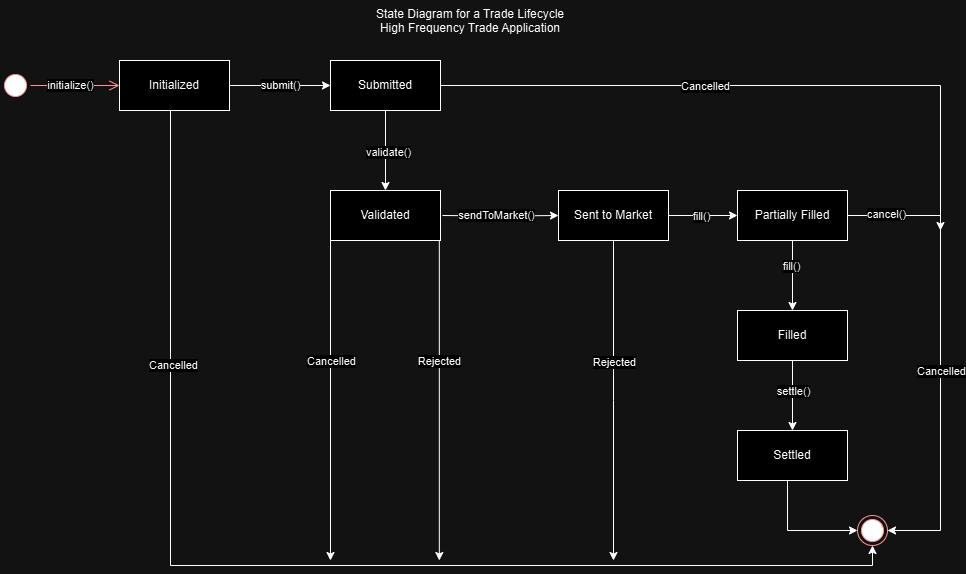




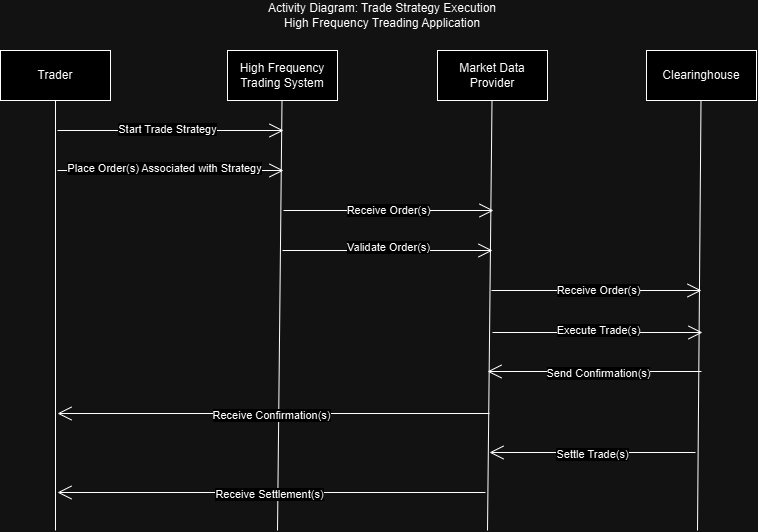


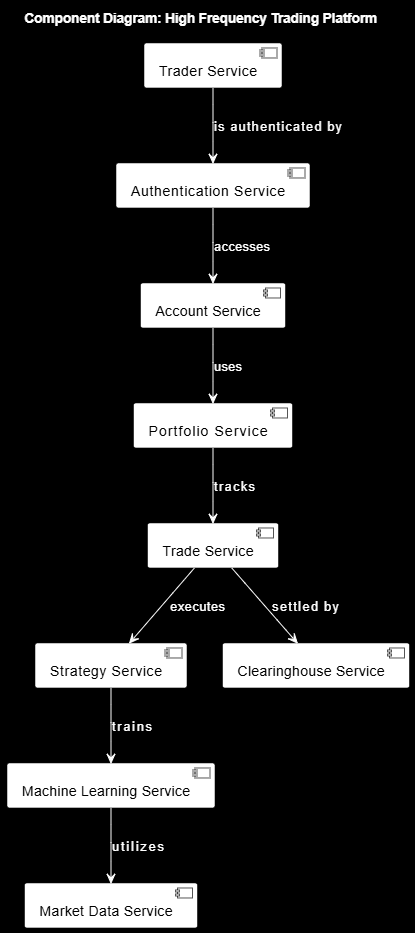
5. UML State Diagram (10 points):

a. Illustrate the possible states and transitions an object can undergo throughout its lifecycle within the system.

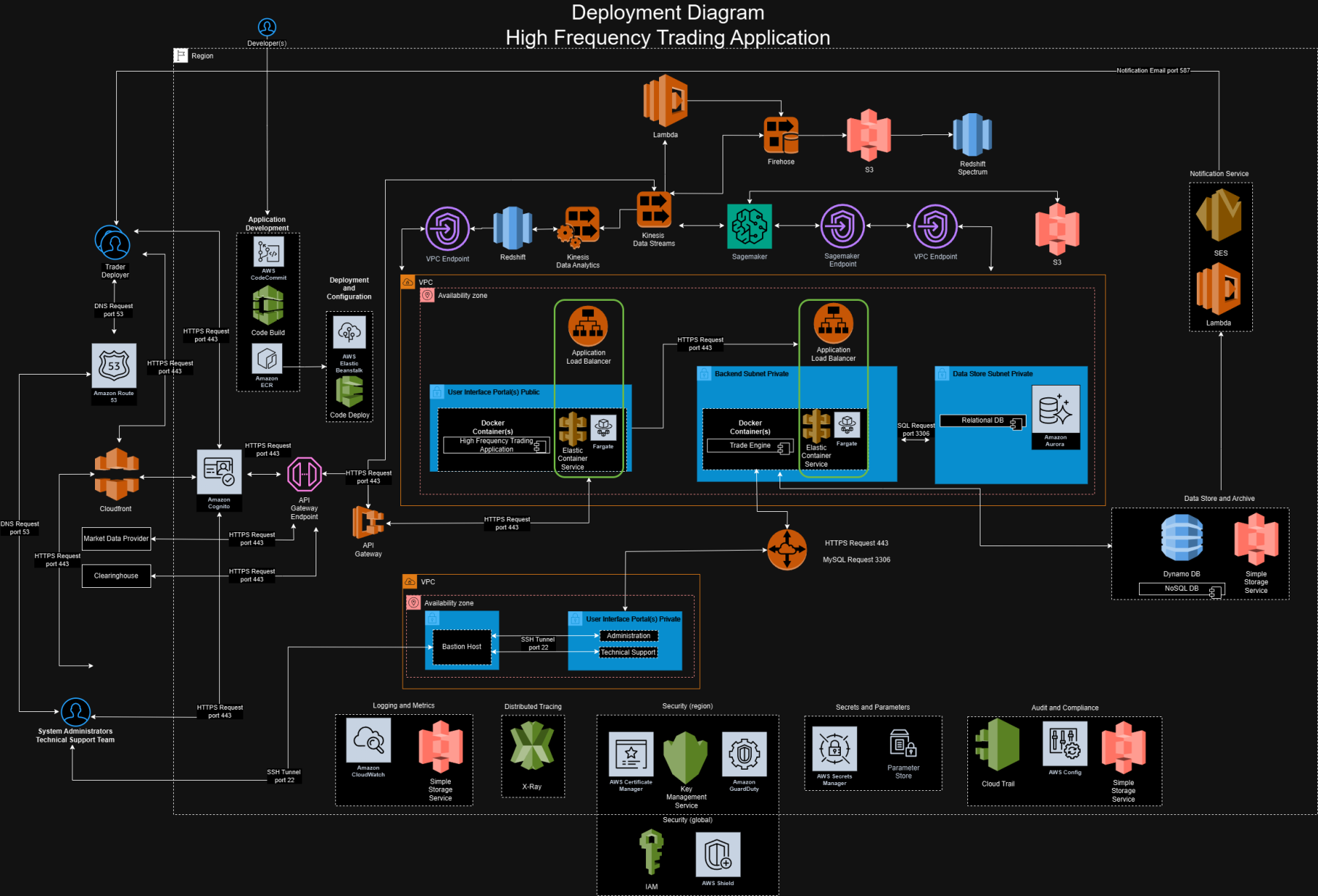


6. UML Activity Diagram (Swimlane Diagram) (10 points):

a. Visually represent the activities and flows within a specific process, highlighting the responsibility of different actors using swim lanes.

7. UML Component Diagram (10 points): a. Depict the system's physical components and their dependencies, providing a high-level architectural view.

8. Cloud Deployment Diagram (10 points):

a. Illustrate the chosen cloud platform (e.g., AWS, Azure) and how the system's components will be deployed within it.

9. Skeleton Classes and Tables Definition (10 points):

a. Provide basic outlines of the main classes involved, including their attributes and methods.

class Account:

def \_\_init\_\_(self, account\_id, account\_type, balance):

self.\_\_account\_id = account\_id

self.\_\_balance = balance

self.\_\_account\_type = account\_type

self.\_\_portfolios = []

@property

def account\_id(self):

return self.\_\_account\_id

@property

def account\_type(self):

return self.\_\_account\_type

@property

def balance(self):

return self.\_\_balance

@property

def portfolios(self):

return self.\_\_portfolios

def add\_portfolio(self, portfolio):

self.\_\_portfolios.append(portfolio)

def deposit(self, amount):

# Code to deposit amount to the account

pass

def get\_balance(self):

# Code to return the account balance

pass

def withdraw(self, amount):

# Code to withdraw amount from the account

pass

class Portfolio:

def \_\_init\_\_(self, account\_id, portfolio\_id, portfolio\_name, portfolio\_type, settled\_cash\_value):

self.\_\_account\_id = account\_id

self.\_\_portfolio\_id = portfolio\_id

self.\_\_portfolio\_name = portfolio\_name

self.\_\_portfolio\_type = portfolio\_type

self.\_\_settled\_cash\_value = settled\_cash\_value

@property

def account\_id(self):

return self.\_\_account\_id

@property

def portfolio\_id(self):

return self.\_\_portfolio\_id

@property

def portfolio\_name(self):

return self.\_\_portfolio\_name

@property

def portfolio\_type(self):

return self.\_\_portfolio\_type

@property

def settled\_cash\_value(self):

return self.\_\_settled\_cash\_value

def add\_order(self, order):

# Code to add an order to the portfolio

pass

def get\_portfolio\_value(self):

# Code to return the portfolio value

pass

def remove\_order(self, order):

# Code to remove an order from the portfolio

pass

class Trader:

def \_\_init\_\_(self, email, name, trader\_id):

self.\_\_email = email

self.\_\_name = name

self.\_\_trader\_id = trader\_id

@property

def email(self):

return self.\_\_email

@property

def name(self):

return self.\_\_name

@property

def trader\_id(self):

return self.\_\_trader\_id

def authenticate(self):

# Code to authenticate the trader

pass

def create\_strategy(self, strategy):

# Code to create a trading strategy

pass

def place\_order(self, order):

# Code to place an order

pass

class Strategy:

def \_\_init\_\_(self, name, parameters, strategy\_id, trader\_id):

self.\_\_name = name

self.\_\_parameters = parameters

self.\_\_strategy\_id = strategy\_id

self.\_\_trader\_id = trader\_id

@property

def name(self):

return self.\_\_name

@property

def parameters(self):

return self.\_\_parameters

@property

def strategy\_id(self):

return self.\_\_strategy\_id

@property

def trader\_id(self):

return self.\_\_trader\_id

def adjust\_parameters(self, new\_parameters):

pass

def apply(self):

# Code to apply the strategy

pass

def evaluate\_performance(self):

# Code to evaluate the performance of the strategy

pass

class Order:

def \_\_init\_\_(self, order\_id, order\_type, price, quantity, status, strategy\_id, symbol, trader\_id, portfolio\_id):

self.\_\_order\_id = order\_id

self.\_\_order\_type = order\_type

self.\_\_price = price

self.\_\_quantity = quantity

self.\_\_status = status

self.\_\_strategy\_id = strategy\_id

self.\_\_symbol = symbol

self.\_\_trader\_id = trader\_id

self.\_\_portfolio\_id = portfolio\_id

def cancel(self):

# Code to cancel the order

pass

def execute(self):

# Code to execute the order

pass

@property

def order\_id(self):

return self.\_\_order\_id

@property

def order\_type(self):

return self.\_\_order\_type

@property

def price(self):

return self.\_\_price

@property

def quantity(self):

return self.\_\_quantity

@property

def status(self):

return self.\_\_status

@property

def strategy\_id(self):

return self.\_\_strategy\_id

@property

def symbol(self):

return self.\_\_symbol

@property

def trader\_id(self):

return self.\_\_trader\_id

@property

def portfolio\_id(self):

return self.\_\_portfolio\_id

def update\_status(self, status):

# Code to update the status of the order

self.\_\_status = status

class MarketData:

def \_\_init\_\_(self, symbol, price, timestamp):

self.\_\_symbol = symbol

self.\_\_price = price

self.\_\_timestamp = timestamp

@property

def symbol(self):

return self.\_\_symbol

@property

def price(self):

return self.\_\_price

@property

def timestamp(self):

return self.\_\_timestamp

def get\_latest\_quote(self):

# Code to get the latest market quote

pass

def subscribe(self, trader):

# Code to subscribe a trader to market data updates

pass

def unsubscribe(self, trader):

# Code to unsubscribe a trader from market data updates

pass

class MachineLearningModel:

def \_\_init\_\_(self, accuracy, model\_id, model\_type, training\_data):

self.\_\_accuracy = accuracy

self.\_\_model\_id = model\_id

self.\_\_model\_type = model\_type

self.\_\_training\_data = training\_data

@property

def accuracy(self):

return self.\_\_accuracy

@property

def model\_id(self):

return self.\_\_model\_id

@property

def model\_type(self):

return self.\_\_model\_type

@property

def training\_data(self):

return self.\_\_training\_data

def evaluate\_accuracy(self):

# Code to evaluate the accuracy of the model

pass

def predict(self, data):

# Code to make a prediction with the model

pass

def train(self):

# Code to train the model

pass

class Clearinghouse:

def \_\_init\_\_(self, clearing\_house\_id, name, location):

self.\_\_clearinghouse\_id = clearinghouse\_id

self.\_\_name = name

self.\_\_location = location

@property

def clearing\_house\_id(self):

return self.\_\_clearing\_house\_id

@property

def name(self):

return self.\_\_name

@property

def location(self):

return self.\_\_location

def settle\_trade(self):

# Code to settle a trade

pass

def report\_status(self):

# Code to report the status of a trade

pass

def validate\_transaction(self):

# Code to validate a transaction

pass

b. Define the structure of any database tables required to store system data.

CREATE TABLE Account (

account\_id INT PRIMARY KEY,

account\_type VARCHAR(50),

balance DECIMAL(15, 2)

);

CREATE TABLE Portfolio (

account\_id INT,

portfolio\_id INT PRIMARY KEY,

portfolio\_name VARCHAR(100),

portfolio\_type VARCHAR(50),

settled\_cash\_value DECIMAL(15, 2),

FOREIGN KEY (account\_id) REFERENCES Account(account\_id)

);

CREATE TABLE Trader (

email VARCHAR(100),

name VARCHAR(100),

trader\_id INT PRIMARY KEY

);

CREATE TABLE Strategy (

name VARCHAR(100),

parameters TEXT,

strategy\_id INT PRIMARY KEY,

trader\_id INT,

FOREIGN KEY (trader\_id) REFERENCES Trader(trader\_id)

);

CREATE TABLE Order (

order\_id INT PRIMARY KEY,

order\_type VARCHAR(10),

price DECIMAL(10, 2),

quantity INT,

status VARCHAR(20),

strategy\_id INT,

symbol VARCHAR(10),

trader\_id INT,

portfolio\_id INT,

FOREIGN KEY (trader\_id) REFERENCES Trader(trader\_id),

FOREIGN KEY (strategy\_id) REFERENCES Strategy(strategy\_id),

FOREIGN KEY (portfolio\_id) REFERENCES Portfolio(portfolio\_id)

);

CREATE TABLE MarketData (

symbol VARCHAR(10),

price DECIMAL(10, 2),

timestamp TIMESTAMP,

PRIMARY KEY (symbol, timestamp)

);

CREATE TABLE MachineLearningModel (

model\_id INT PRIMARY KEY,

model\_type VARCHAR(50),

accuracy DECIMAL(5, 2),

training\_data TEXT

);

CREATE TABLE Clearinghouse (

clearinghouse\_id INT PRIMARY KEY,

name VARCHAR(100),

location VARCHAR(100)

);

10.Design Patterns (10 points):

a. Explain any GRASP, SOLID, GOF, Microservices design patterns and best practices implemented in your design, justifying their use for specific scenarios.

**Design Patterns and Best Practices Implemented in a High-Frequency Trading Platform**

The design and implementation of a high-frequency trading platform required careful consideration of best practices and design patterns to ensure robustness, maintainability and scalability. The following key principles and patterns have been applied in the system design to achieve these goals:

**YAGNI (You Aren't Gonna Need It)**

**Implementation:**  
The platform is designed to avoid unnecessary features or functionalities that are not currently required. For instance, the Account class includes only the basic attributes and methods essential for managing an account, excluding any additional features that are not immediately needed.

**Justification:**  
This approach prevents over engineering, keeping the system lean and focused on current requirements. It simplifies maintenance and enhances the clarity of the codebase, making it easier for developers to understand and modify as necessary.

**Database Normalization Techniques and DRY Principle**

Database normalization is a crucial process in relational database design aimed at minimizing redundancy and dependency by organizing fields and table structures according to certain normal forms. The table definitions provided reflect several levels of normalization, contributing to the efficiency, consistency and integrity of the database. Additionally, the principles of DRY (Don't Repeat Yourself) have been integrated into the design, further enhancing the maintainability and robustness of the system.

**First Normal Form (1NF)**

The first normal form requires that the data is stored in a table format where each column contains atomic values and each entry in a column is of the same data type. Additionally, each table must have a primary key that uniquely identifies each row.

In the provided table definitions, all tables adhere to the 1NF requirements:

* **Account, Portfolio, Trader, Strategy, Order, MarketData, MachineLearningModel and Clearinghouse** tables each contain atomic values in their columns. For instance, the Account table includes columns such as account\_id, account\_type and balance, where each column holds a single piece of information.
* Each table has a clearly defined primary key, such as account\_id for the Account table, portfolio\_id for the Portfolio table and so on, ensuring that each record is uniquely identifiable.

**Second Normal Form (2NF)**

The second normal form builds on 1NF by ensuring that all non-key attributes are fully functionally dependent on the entire primary key. This form eliminates partial dependency, where a non-key attribute depends only on a part of the primary key.

In these table definitions:

* **Portfolio**: The Portfolio table is in 2NF because its non-key attributes (portfolio\_name, portfolio\_type, settled\_cash\_value) depend fully on the primary key portfolio\_id. The account\_id acts as a foreign key linking back to the Account table, which avoids partial dependency on the portfolio\_id.
* **Order**: The Order table's attributes such as order\_type, price, quantity and status all depend on the primary key order\_id, adhering to 2NF. The inclusion of foreign keys (strategy\_id, trader\_id and portfolio\_id) links the Order table to other entities but does not introduce partial dependencies on the composite keys.

**Third Normal Form (3NF)**

The third normal form eliminates transitive dependencies, where non-key attributes are dependent on other non-key attributes rather than the primary key.

In these table structures:

* **Trader and Strategy**: The Trader table holds independent attributes (email, name, trader\_id) and the Strategy table, which links to the Trader table via trader\_id, also adheres to 3NF. The attributes name and parameters in the Strategy table are dependent only on the primary key strategy\_id.
* **MarketData**: The MarketData table's structure ensures 3NF as the non-key attributes (price, timestamp) depend directly on the composite primary key (symbol, timestamp), with no transitive dependencies.

**Considerations for Further Normalization (Beyond 3NF)**

In a well-normalized database, sometimes denormalization is considered for performance reasons. However, given the critical nature of financial and trading systems, further normalization or maintaining the current structure might be necessary to ensure data integrity and consistency.

For example:

* **Strategy Table**: If the parameters attribute in the Strategy table contains multiple, potentially complex pieces of data (e.g., JSON configurations), further normalization might involve decomposing this into a separate table to avoid potential anomalies and to maintain clarity.
* **Order Table**: The Order table might be subjected to further scrutiny if the status attribute could involve complex state management, which might suggest the creation of an OrderStatus table to manage status states and transitions more formally.

**DRY (Don't Repeat Yourself) Principle**

**Implementation:**  
The DRY principle has been effectively implemented in the design of the database tables. Common functionalities and related attributes are centralized within specific tables, eliminating redundancy. For example, the Trader, Account and Portfolio tables encapsulate related attributes and methods, ensuring that all functionality related to these entities is consistently managed in one place.

**Justification:**  
By adhering to the DRY principle, the design reduces the risk of inconsistencies within the database structure. For instance, if the logic or rules associated with placing an order need to be updated, changes can be made in a single location, such as the Order table, without the need to modify multiple tables. This approach minimizes errors, enhances maintainability and ensures that the database remains easy to manage and extend.

**Referential Integrity and Foreign Keys**

The use of foreign keys in these table definitions enforces referential integrity across the database:

* **Portfolio Table**: The foreign key account\_id ensures that every portfolio is linked to an existing account in the Account table.
* **Order Table**: Foreign keys like trader\_id, strategy\_id and portfolio\_id ensure that orders are tied to valid traders, strategies and portfolios, respectively.

By employing these foreign keys, the database ensures that relationships between tables are consistent and that orphaned records are prevented, enhancing the reliability of the data.

The database tables provided adhere to the principles of normalization up to at least the third normal form (3NF), ensuring that the database structure is optimized for consistency, integrity and efficiency. Furthermore, the implementation of the DRY principle within the database design reduces redundancy, minimizes the likelihood of inconsistencies and makes the system easier to maintain. While these tables are well-designed to minimize redundancy and ensure data integrity, further normalization or careful consideration of denormalization may be necessary depending on specific performance needs and the complexity of data relationships within the trading platform.

**KISS (Keep It Simple, Stupid)**

**Implementation:**  
The system's design prioritizes simplicity and clarity. Each class is assigned a clear responsibility with a limited scope and methods such as deposit, withdraw, add\_order and execute are designed to perform single, well-defined tasks.

**Justification:**  
Simplicity in design ensures that the system is easy to understand, maintain and extend. It reduces the potential for errors and makes the code more accessible to new developers, fostering a more maintainable and adaptable codebase.

**GRASP (General Responsibility Assignment Software Patterns)**

**Implementation:**

* **Information Expert:** Each class manages its own data and logic, such as the Account class handling account-related operations and the Order class managing order-specific operations.
* **Creator:** Classes that use other objects are responsible for creating them, as seen in the Trader class, which creates Strategy and Order objects.
* **Controller:** A central controller, such as the Trading Engine, manages the flow of activities and coordinates the interactions between different components.

**Justification:**  
GRASP patterns help distribute responsibilities appropriately across the system, resulting in a modular and cohesive design. This structure supports easier maintenance and scalability by ensuring that each component has a clear and focused role.

**SOLID Principles**

**Implementation:**

* **Single Responsibility Principle (SRP):** Each class has a single responsibility, with the Order class, for example, focusing exclusively on order-related operations.
* **Open/Closed Principle (OCP):** Classes are designed to be open for extension but closed for modification, allowing for new functionality to be added without altering existing code.
* **Dependency Inversion Principle (DIP):** Higher-level modules rely on abstractions rather than lower-level modules, as exemplified by the Trader class using Strategy and Order objects without needing to know their internal implementations.

**Justification:**  
The SOLID principles ensure that the system remains modular, extensible and easy to maintain. These principles promote the long-term sustainability of the codebase, allowing for the smooth integration of new features and reducing the risk of introducing errors during updates.

**GOF (Gang of Four) Design Patterns**

**Implementation:**

* **Strategy Pattern:** The Strategy class encapsulates different trading strategies, which can be dynamically applied by the Trader class.
* **Observer Pattern:** The MarketData class can implement an observer pattern to notify subscribed traders of market data updates in real-time.

**Justification:**  
These design patterns provide tried-and-true solutions to common design challenges. The strategy pattern, for example, allows for the flexible implementation of various trading strategies, while the observer pattern facilitates real-time updates, enabling traders to respond swiftly to market changes.

**Microservices Design Patterns**

**Implementation:**  
The system is designed to be decomposable into microservices, such as Trader Service, Order Service and Market Data Service. Each microservice is responsible for a specific aspect of the trading platform and communicates with other services through APIs.

**Justification:**  
Microservices architecture offers several advantages, including the independent development, deployment and scaling of different components. This flexibility enhances the system's resilience and allows it to adapt quickly to changing requirements.

**Specific Scenarios Justifying the Use of Design Patterns**

**High Availability and Scalability:**  
By leveraging microservices, individual components can be scaled independently based on their load. For example, during peak trading hours, the Order Service can be scaled up without impacting other services.

**Maintainability and Extensibility:**  
The adherence to SOLID principles ensures that new features or updates can be incorporated with minimal disruption to existing code. For instance, a new trading strategy can be added by extending the Strategy class without requiring changes to the Trader class.

**Real-time Data Processing:**  
The observer pattern used in the MarketData class enables efficient handling of real-time market data updates, ensuring that traders are promptly notified as new data becomes available.

**Security and Data Integrity:**  
By following YAGNI and KISS principles, only necessary features are implemented, reducing the system's attack surface and potential vulnerabilities. Moreover, each class is responsible for managing its own data, ensuring data integrity and consistency throughout the system.

By implementing these design patterns and best practices, the high-frequency trading platform is designed to be robust, flexible and maintainable. The system's architecture supports efficient real-time trading, scalability and adaptability to future requirements, ensuring long-term sustainability and performance in a demanding trading environment.

**WORK CITED**

Ambler, Scott W. "Data Normalization: Beyond 3rd Normal Form (3NF)." Agile Data, [www.agiledata.org/essays/dataNormalization.html](http://www.agiledata.org/essays/dataNormalization.html). Accessed 10 Aug. 2024.

Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd ed., Pearson, 2004.