Problem 2 : Build a real-time alerting system

Algorithm and Data Structures: To design the real-time alerting system for monitoring transaction activities, the following algorithm and data structures can be utilized:

Data Ingestion: Use a streaming mechanism like Apache Kafka to ingest transaction data in real-time.

Data Storage: Store the transactions in an optimized time-series database like InfluxDB, TimescaleDB, or Apache Cassandra. These databases are designed to handle high volume writes and provide efficient querying capabilities.

Threshold-based Alert:

Maintain a threshold value for transaction amounts.
For each incoming transaction, compare its amount against the threshold value.
If the transaction amount exceeds the threshold, trigger an alert.

Rate-based Alert:

Maintain a sliding time window (e.g., last hour) for monitoring transaction rates.
Calculate the average transaction rate within the window.
Monitor the incoming transactions and count the number of transactions within the window.
If the transaction rate exceeds a certain multiple (e.g., 10x) of the average rate, trigger an alert.

Alert Triggering:

Implement a notification system to trigger alerts to fund managers.
Use communication channels like email, SMS, or push notifications to notify managers about the alerts.
Alert Management:

Maintain a log or database to track alerts.
Include functionalities to acknowledge, escalate, or resolve alerts.

Pseudocode:

```
threshold = predefined_threshold_value // set the threshold for transaction amount
window_size = 1 hour // set the time window for rate-based alert
rate_threshold_factor = 10 // factor to determine the spike in the transaction rate

transaction_stream.subscribe {
```

```
    foreach transaction in transaction_stream {
        handleThresholdBasedAlert(transaction)
        handleRateBasedAlert(transaction)
    }
}

function handleThresholdBasedAlert(transaction) {
    if transaction.amount > threshold {
        triggerAlert(transaction, "Threshold-based Alert")
    }
}

function handleRateBasedAlert(transaction) {
    transactions_in_window = getTransactionsWithinWindow(transaction.timestamp -
window_size, transaction.timestamp)
    average_rate = transactions_in_window.length / window_size
    transaction_rate = countTransactionsWithinWindow(transactions_in_window)

    if transaction_rate > (average_rate * rate_threshold_factor) {
        triggerAlert(transaction, "Rate-based Alert")
    }
}

function triggerAlert(transaction, alert_type) {
    alert = createAlert(transaction, alert_type)
    sendAlertToManagers(alert)
}

function createAlert(transaction, alert_type) {
    // Create an alert object with relevant information
    alert = {
        transaction_id: transaction.id,
        transaction_amount: transaction.amount,
        alert_type: alert_type,
        timestamp: current_timestamp
    }
    return alert
}

function sendAlertToManagers(alert) {
    // Send the alert notification to fund managers using the chosen communication channels
    // Example: send email, SMS, or push notification to the designated managers
    // Implement the logic specific to the chosen communication channels
}
```

Scalability, Data Integrity, and Fault Tolerance:

Scalability: The system can scale horizontally by partitioning the transaction data across multiple instances of the real-time processing components such as Kafka consumers. This ensures that the system can handle a high volume of transactions by distributing the processing load.
Data Integrity: To ensure data integrity, transaction data can be stored in a fault-tolerant and durable database system, like Cassandra or a distributed file system. Additionally, the system can employ data replication and backup strategies to minimize the risk of data loss or corruption.
Fault Tolerance: The system can incorporate fault-tolerant mechanisms such as Kafka's replication and fault recovery capabilities. By deploying Kafka in a clustered mode with proper configurations, the system can withstand failures of individual components and maintain data consistency.
Evaluation Criteria:

Clarity and Efficiency of the Algorithm: Evaluate the algorithm's clarity and how effectively it fulfills the requirements.

Scalability and Reliability Considerations: Examine the system's ability to scale horizontally and handle high transaction volumes. Assess the strategies for data integrity and fault tolerance.

Error Handling, Validation, and Robustness: Evaluate how the system handles errors, performs data validation, and maintains robustness under different scenarios.
Understanding of Real-Time Processing Challenges and Solutions: Assess the understanding of real-time processing challenges and the proposed solutions.