**Description of System 1**: We are looking for a financial system that monitors investments. The requirements are well defined. The system needs to be available 100% of the time. Our clients are our priority, their data need to remain private and we need to maintain their integrity.

**Functional Requirement:**

1. Monitor Investments: The system shall monitor clients' investments.
2. Client Data Management: The system shall store, retrieve, and update clients' investment data.
3. Client Interaction: The system shall allow clients to view their investment data in real-time.
4. Alerts and Notifications: The system shall notify clients of significant changes in their investments (e.g., value fluctuations).
5. Transaction Processing: The system shall allow clients to perform investment-related transactions (e.g., buying, selling, transferring funds).
6. Reporting: The system shall generate detailed reports of client investments over specified periods.

**Non-Functional Requirements:**

1. Availability: The system shall be available 100% of the time.
2. Security: The system shall ensure that client data remains private and protected against unauthorized access.
3. Data Integrity: The system shall maintain the integrity of client data, ensuring accuracy and consistency across transactions.
4. Performance: The system shall process transactions and update investment data in real-time, with response times of less than 1 second.
5. Scalability: The system shall scale to accommodate an increasing number of clients and investment data without performance degradation.
6. Compliance: The system shall comply with all relevant financial regulations and standards.
7. Backup and Recovery: The system shall have mechanisms for automatic data backup and fast recovery in case of failure, with a maximum recovery time objective (RTO) of 5 minutes.
8. Auditability: The system shall maintain detailed logs of all transactions and changes to client data for auditing purposes.

**Use case Scenarios:**

**Use Case 1: Monitor Client Investments**

* **Actors**: Client, System
* **Preconditions**: Client is logged into the system.
* **Trigger**: Client requests to view their current investments.
* **Main Success Scenario**:
  1. Client accesses the "Investments" section of the system.
  2. The system retrieves the latest investment data for the client.
  3. The system displays the current value of the client's investments.
* **Postconditions**: Client views up-to-date information on their investments.

**Use Case 2: Receive Investment Notifications**

* **Actors**: System, Client
* **Preconditions**: Client is subscribed to notifications, and investment monitoring is active.
* **Trigger**: The value of a client's investment changes significantly.
* **Main Success Scenario**:
  1. The system detects a significant change in the client's investment value.
  2. The system generates a notification with the details of the change.
  3. The system sends the notification to the client (via email, SMS, or in-app alert).
* **Postconditions**: Client receives a notification about the investment change.

**Use Case 3: Perform Investment Transaction**

* **Actors**: Client, System, Financial Institution
* **Preconditions**: Client is logged into the system, and funds are available for transaction.
* **Trigger**: Client initiates an investment transaction (e.g., buy or sell).
* **Main Success Scenario**:
  1. Client selects the desired transaction type (buy/sell) and specifies the amount.
  2. The system validates the transaction details and available funds.
  3. The system communicates with the financial institution to execute the transaction.
  4. The system updates the client's investment data to reflect the transaction.
  5. The system confirms the transaction with the client.
* **Postconditions**: The investment transaction is completed, and the client receives confirmation.

**Use Case 4: Generate Investment Report**

* **Actors**: Client, System
* **Preconditions**: Client is logged into the system.
* **Trigger**: Client requests a report on their investment performance over a specific period.
* **Main Success Scenario**:
  1. Client navigates to the "Reports" section of the system.
  2. Client selects the desired report type and time period.
  3. The system retrieves and compiles the relevant investment data.
  4. The system generates the investment report in the requested format (e.g., PDF, CSV).
  5. The system provides the report to the client for download or viewing.
* **Postconditions**: Client receives the requested investment report.

**Use Case 5: Update Client Investment Data**

* **Actors**: Client, System
* **Preconditions**: Client is logged into the system.
* **Trigger**: Client initiates an update to their personal information or investment preferences.
* **Main Success Scenario**:
  1. Client navigates to the "Account Settings" or "Investment Preferences" section of the system.
  2. Client updates their information (e.g., contact details, risk tolerance).
  3. The system validates the updated information.
  4. The system saves the updated information to the client's profile.
  5. The system confirms the update with the client.
* **Postconditions**: Client information is updated successfully.

**Domain concepts:**

**1. Client**

* A person or entity using the financial system to manage and monitor investments.
* Attributes: Personal Information (Name, Contact Details), Account Information, Investment Preferences, Risk Tolerance.

2.**Investment**

* Financial assets held by the client, such as stocks, bonds, mutual funds, or other securities.
* Attributes: Type of Investment, Value, Purchase Date, Performance, Risk Level.

**3. Transaction**

* An action performed by the client to buy, sell, or transfer investments.
* Attributes: Transaction Type (Buy, Sell, Transfer), Amount, Date, Status, Related Investment.

**4. Notification**

* Alerts or messages sent to the client regarding significant changes in their investments or system updates.
* Attributes: Notification Type (Price Alert, System Alert), Content, Delivery Method (Email, SMS, In-App), Timestamp.

**5. Report**

* A summary or detailed document generated by the system that provides insights into the performance of a client's investments.
* Attributes: Report Type (Summary, Detailed, Period-Based), Format (PDF, CSV), Date Range, Content.

**6.Investment Data**

* The collective information about a client's investments, including historical data, current values, and performance metrics.
* Attributes: Historical Data, Current Value, Performance Metrics, Market Data.

**7.Security**

* Measures and mechanisms put in place to protect client data and ensure its privacy and confidentiality.
* Attributes: Encryption, Authentication Methods, Access Control, Data Masking.

**8.Data Integrity**

* Ensuring that client data is accurate, consistent, and protected from unauthorized modifications.
* Attributes: Validation Rules, Data Consistency, Audit Trails.

**9.Availability**

* The system's ability to be operational and accessible to clients 100% of the time.
* Attributes: Uptime, Redundancy, Failover Mechanisms.

**10.Scalability**

* The system's capability to handle an increasing number of clients and transactions without degradation in performance.
* Attributes: Load Balancing, Elastic Resources, Performance Monitoring.

**11.Compliance**

* Adherence to financial regulations and standards that govern the handling of client data and transactions.
* Attributes: Regulatory Requirements, Auditing, Reporting Standards.

**12.Backup and Recovery**

* The processes involved in ensuring that client data is backed up and can be recovered in case of a system failure.
* Attributes: Backup Frequency, Recovery Time Objective (RTO), Recovery Procedures.

**13.Audit Log**

* A record of all actions and transactions performed within the system for auditing and compliance purposes.
* Attributes: Action Type, User, Timestamp, Details of Change.

**14.Financial Institution**

* External entities that the system interacts with to execute investment transactions on behalf of clients.
* Attributes: Institution Name, Transaction Protocols, Integration Methods.

**15.Client Preferences**

* Configurable options that allow clients to customize their interaction with the system, such as notification settings and investment strategies.
* Attributes: Notification Preferences, Investment Strategy, Risk Profile.

**16.Account Settings**

* The area where clients manage their personal and account-related information.
* Attributes: Contact Information, Password, Security Questions, Linked Accounts.

**Domain Model:**

Picture attached domain\_model\_1

**Suggested Architectural Style**: Microservices Architecture

Given the domain model and the requirements for high availability, scalability, security, and modular functionality, the **Microservices Architecture** is a strong candidate for implementing this system. Microservices allow the system to be broken down into small, independently deployable services that can each handle a specific domain concept or group of related concepts.

**Key Advantages of Microservices for This Domain:**

* **Scalability**: Individual services (e.g., TransactionService, NotificationService) can scale independently based on demand.
* **Availability**: The system can maintain availability even if one service fails, thanks to the distributed nature of microservices.
* **Security**: Security can be managed at the service level, enforcing strict access controls and handling sensitive data in isolation.
* **Flexibility**: Each microservice can use the best technology for its specific task, allowing for flexibility in implementation and optimization.

Diagram attached as component\_diagram\_1\_1

**Justification for the architecture:**

The Microservices Architecture is well-suited for implementing the financial system described for several key reasons:

1. High Availability:
   * Requirement: The system must be available 100% of the time.
   * Justification: Microservices architecture allows for distributed deployment across multiple servers and locations, ensuring that the failure of one service does not impact the entire system. Each service can be monitored and restarted independently, increasing overall system uptime.
2. Scalability:
   * Requirement: The system must scale to accommodate growing numbers of clients and data without performance degradation.
   * Justification: Microservices support horizontal scaling, where individual services can scale independently based on their specific needs. For example, if transaction volumes increase, only the TransactionService needs to be scaled, rather than the entire system. This optimizes resource usage and ensures consistent performance under varying loads.
3. Security:
   * Requirement: Client data must remain private and secure.
   * Justification: Microservices allow for security to be implemented at the service level. Sensitive services (e.g., SecurityService, ClientService) can have stricter access controls, encryption, and authentication mechanisms. This isolation limits the impact of potential security breaches to specific services, enhancing overall system security.
4. Modularity and Flexibility:
   * Requirement: The system needs to be maintainable and able to evolve as new requirements arise.
   * Justification: Microservices architecture promotes modularity, where each service is responsible for a specific business function. This makes the system easier to understand, develop, and maintain. New features or changes can be made to individual services without impacting the entire system. Additionally, different technologies and frameworks can be used for different services, allowing for flexibility in implementation.
5. Resilience and Fault Isolation:
   * Requirement: The system should handle failures gracefully without impacting the entire system.
   * Justification: In microservices, faults in one service are isolated to that service, and not the entire application. For instance, if the NotificationService fails, other core functionalities like TransactionService or InvestmentService remain unaffected. This fault isolation improves overall system resilience.
6. Ease of Continuous Deployment:
   * Requirement: The system may require frequent updates and improvements.
   * Justification: Microservices architecture enables continuous deployment and integration. Since services are decoupled, updates to one service can be deployed independently without needing to redeploy the entire system. This reduces downtime and allows for faster iteration and delivery of new features or bug fixes.
7. Compliance and Auditability:
   * Requirement: The system must comply with financial regulations and maintain audit trails.
   * Justification: Microservices facilitate the implementation of compliance and audit services as separate, dedicated components (e.g., ComplianceService, AuditLogService). This ensures that regulatory requirements are met without affecting the core functionalities of the system. Additionally, different jurisdictions can be handled by specific microservices that cater to localized regulations.
8. Efficient Resource Utilization:
   * Requirement: The system should manage resources efficiently to handle varying loads.
   * Justification: Microservices allow for dynamic allocation of resources based on the demand of each service. For example, the TransactionService might require more resources during market hours, while ReportService may need more resources at the end of the day. This flexibility ensures efficient utilization of system resources.

Conclusion:

The Microservices Architecture is justified for this financial system due to its ability to meet the critical requirements of availability, scalability, security, modularity, and compliance. It provides the necessary flexibility and resilience to build a robust financial monitoring system that can evolve with changing business needs while ensuring continuous and reliable service delivery.

**Suggested Architectural Style**: Event Driven Architecture

An Event-Driven Architecture (EDA) can also be a viable option for implementing the financial system. In this architecture, components or services communicate with each other through events. The system reacts to changes in state (events) and triggers appropriate responses, making it highly flexible and scalable. This approach is particularly suitable for systems that need to respond to asynchronous events, such as financial transactions, notifications, and data updates.

Key Advantages of Event-Driven Architecture for This Domain:

1. Asynchronous Communication: EDA allows different parts of the system to communicate asynchronously, which is ideal for real-time updates and notifications without blocking processes.
2. Loose Coupling: Services in EDA are loosely coupled, which enhances flexibility and scalability. Each service only needs to subscribe to or publish relevant events, reducing dependencies.
3. Scalability: EDA allows for scalable event processing, making it easier to handle large volumes of transactions and notifications, which is crucial for financial systems.
4. Resilience: EDA inherently supports fault tolerance by decoupling services. If one component fails, other components can continue to operate as long as they are not directly affected by the failure.
5. Real-Time Processing: EDA can handle real-time data processing, making it suitable for investment monitoring, notifications, and quick responses to financial market changes.

Diagram attached as component\_diagram\_1\_2

**Justification for Using Event-Driven Architecture (EDA)**

An **Event-Driven Architecture (EDA)** is a suitable architectural style for the financial system due to the following reasons:

1. **Real-Time Responsiveness**:
   * The financial system needs to handle real-time events, such as investment changes, market fluctuations, and transaction updates. EDA excels in scenarios where the system must respond immediately to changes. Events are processed as they happen, enabling instant notifications and real-time data updates for clients.
2. **Asynchronous Communication**:
   * The system components can communicate asynchronously, which means that processes do not need to wait for each other to complete before continuing their operations. For example, when a transaction is initiated, the TransactionService can publish an event, and other services (e.g., InvestmentService, NotificationService) can react to that event without blocking. This leads to a more efficient and scalable system.
3. **Loose Coupling**:
   * EDA promotes loose coupling between services. Each service only needs to know about the events it consumes or produces, without being tightly integrated with other services. This reduces dependencies and allows services to evolve independently. For instance, changes to the InvestmentService do not directly affect the NotificationService, as long as the event structure remains consistent.
4. **Scalability**:
   * EDA allows individual services to scale independently based on the volume of events they handle. For example, during market fluctuations, the TransactionEventQueue can scale to handle a higher volume of buy/sell events, while other services, such as the ReportService, can remain unaffected. Event queues can be scaled horizontally, handling high throughput without bottlenecks.
5. **Resilience and Fault Tolerance**:
   * EDA enhances the resilience of the system by decoupling services and relying on an event-driven communication model. If one service fails (e.g., the NotificationService), other services can continue to operate, and events can be queued until the service recovers. This architecture ensures that a failure in one component does not bring down the entire system.
   * Events can also be persisted in the event queue, ensuring that no critical data is lost, even in the case of temporary service outages.
6. **Flexibility and Extensibility**:
   * EDA allows for easy integration of new services. As the financial system evolves, new features (e.g., a new reporting service or advanced analytics) can be added by simply subscribing to existing events or publishing new ones. This flexibility makes it easier to adapt to changing business requirements without requiring a complete system overhaul.
   * Services can be developed, deployed, and tested independently, making it easier to manage ongoing system improvements.
7. **Event Sourcing and CQRS**:
   * EDA can easily incorporate advanced patterns such as **Event Sourcing** and **Command Query Responsibility Segregation (CQRS)**. Event sourcing allows the system to maintain a full history of changes to data by storing events rather than the current state. This is useful for auditability and compliance, ensuring that all actions are traceable.
   * CQRS allows the separation of read and write operations, optimizing the system for performance and scalability.
8. **Real-Time Notifications**:
   * EDA is ideal for handling real-time notifications. For example, when an investment's value changes significantly, an event is triggered, and the NotificationService can immediately notify the client. The decoupling provided by EDA ensures that notifications are handled efficiently without impacting the core investment processing services.
9. **Improved Performance**:
   * EDA can optimize performance by distributing the load across multiple event handlers. When events are processed asynchronously, the system can handle a large number of transactions and updates concurrently, improving overall throughput.

**Conclusion**

Event-Driven Architecture is an excellent choice for implementing this financial system due to its real-time responsiveness, asynchronous communication, and ability to scale and respond to events efficiently. It provides the necessary flexibility, fault tolerance, and decoupling required for a complex system handling sensitive financial data. Moreover, it enables easy integration of new services, making it a future-proof architecture that can evolve with changing business needs.

**Software Architecture Analysis Method (SAAM) for Microservices Architecture**

The Software Architecture Analysis Method (SAAM) is a method used to evaluate software architectures, particularly their modifiability, performance, and other quality attributes. To apply SAAM to the proposed microservices architecture for the financial system, we need to identify and document system scenarios that will help evaluate the architecture's ability to meet the functional and non-functional requirements.

**Step 1: Identify the Stakeholders**

Stakeholders for this financial system include:

* **Clients**: Individuals or entities using the system to manage their investments.
* **System Administrators**: Responsible for maintaining the system's availability, security, and performance.
* **Developers**: Responsible for implementing, maintaining, and updating the system.
* **Compliance Officers**: Ensuring that the system adheres to financial regulations.
* **Financial Institutions**: External entities involved in executing transactions.

**Step 2: Define the System Scenarios**

Scenarios in SAAM are used to understand how the architecture handles various use cases and requirements. We will define scenarios in three categories: **use cases**, **modification scenarios**, and **failure scenarios**.

**1. Use Case Scenarios**

These scenarios describe normal operations of the system and represent its main functionality.

* **Scenario 1: Monitoring Investments**
  + **Description**: A client logs in to the system and views their current investment portfolio, including real-time data on their investments.
  + **Quality Attributes**: Availability, Performance.
  + **Relevant Services**: ClientService, InvestmentService, InvestmentDataService, NotificationService.
* **Scenario 2: Executing a Transaction**
  + **Description**: A client initiates a transaction to buy or sell stocks. The system processes the transaction and updates the client's portfolio.
  + **Quality Attributes**: Performance, Data Integrity.
  + **Relevant Services**: ClientService, TransactionService, FinancialInstitutionService, InvestmentDataService, AuditLogService.
* **Scenario 3: Receiving Notifications**
  + **Description**: A client receives a notification when a significant change occurs in their investment portfolio.
  + **Quality Attributes**: Responsiveness, Security.
  + **Relevant Services**: NotificationService, ClientService, InvestmentService.
* **Scenario 4: Generating Reports**
  + **Description**: A client requests a report on their investment performance over the past year, and the system generates the report.
  + **Quality Attributes**: Performance, Availability.
  + **Relevant Services**: ReportService, InvestmentDataService, TransactionService.

**2. Modification Scenarios**

These scenarios describe changes to the system, evaluating how the architecture handles modifications and updates.

* **Scenario 5: Adding a New Investment Type**
  + **Description**: The system needs to support a new type of investment, such as cryptocurrency. This requires updating the InvestmentService, InvestmentDataService, and TransactionService.
  + **Quality Attributes**: Modifiability, Scalability.
  + **Relevant Services**: InvestmentService, TransactionService, InvestmentDataService.
* **Scenario 6: Enhancing Security Protocols**
  + **Description**: The system's security protocols need to be updated to comply with new regulations. This involves changes to the SecurityService and the authentication mechanism across all services.
  + **Quality Attributes**: Security, Modifiability.
  + **Relevant Services**: SecurityService, ClientService, TransactionService.
* **Scenario 7: Scaling Up for Increased Load**
  + **Description**: Due to an increase in clients, the system must scale to handle a higher volume of transactions and data processing.
  + **Quality Attributes**: Scalability, Performance.
  + **Relevant Services**: TransactionService, InvestmentService, ScalabilityService, AvailabilityService.
* **Scenario 8: Integrating with a New Financial Institution**
  + **Description**: The system needs to integrate with a new external financial institution to process transactions. This requires changes to the FinancialInstitutionService and TransactionService.
  + **Quality Attributes**: Modifiability, Availability.
  + **Relevant Services**: FinancialInstitutionService, TransactionService.

**3. Failure Scenarios**

These scenarios explore how the system handles failures, ensuring that it meets the availability, fault tolerance, and recovery requirements.

* **Scenario 9: Service Failure in NotificationService**
  + **Description**: The NotificationService fails during a high-volume event. The system must ensure that other services (e.g., TransactionService, InvestmentService) continue operating normally.
  + **Quality Attributes**: Fault Tolerance, Availability.
  + **Relevant Services**: NotificationService, TransactionService, InvestmentService.
* **Scenario 10: Data Loss in InvestmentDataService**
  + **Description**: The InvestmentDataService experiences data loss due to a hardware failure. The system must recover the data from backups and restore operations with minimal downtime.
  + **Quality Attributes**: Backup and Recovery, Data Integrity.
  + **Relevant Services**: InvestmentDataService, BackupAndRecoveryService.
* **Scenario 11: Latency in Transaction Processing**
  + **Description**: Clients experience high latency during peak hours when processing transactions. The system must handle load balancing to improve response times.
  + **Quality Attributes**: Performance, Scalability.
  + **Relevant Services**: TransactionService, ScalabilityService, AvailabilityService.
* **Scenario 12: Security Breach in ClientService**
  + **Description**: A security breach is detected in the ClientService. The system must isolate the breach, protect other services, and notify the relevant stakeholders.
  + **Quality Attributes**: Security, Fault Tolerance.
  + **Relevant Services**: SecurityService, ClientService, AuditLogService.

**Step 3: Evaluate the Architecture Against the Scenarios**

To evaluate the microservices architecture using SAAM, you would analyze how well the architecture handles each scenario in terms of the relevant quality attributes (e.g., modifiability, scalability, fault tolerance). Specifically, you would assess:

1. **Interactions Between Services**: How services communicate and handle dependencies, ensuring that failures in one service do not cascade to others.
2. **Modifiability**: The ease with which the system can be updated or extended to support new functionality or meet new requirements.
3. **Performance and Scalability**: How well the system scales under increased load and maintains performance.
4. **Security and Fault Tolerance**: How the system handles security breaches and service failures to maintain data integrity and system availability.
5. **Backup and Recovery**: The system's ability to recover from data loss or service outages with minimal impact on clients.

**Conclusion**

By documenting and evaluating these scenarios, SAAM allows us to understand how well the microservices architecture supports the core requirements of the financial system, including availability, scalability, security, and modifiability. This structured evaluation helps ensure that the architecture is robust and can handle real-world challenges effectively.