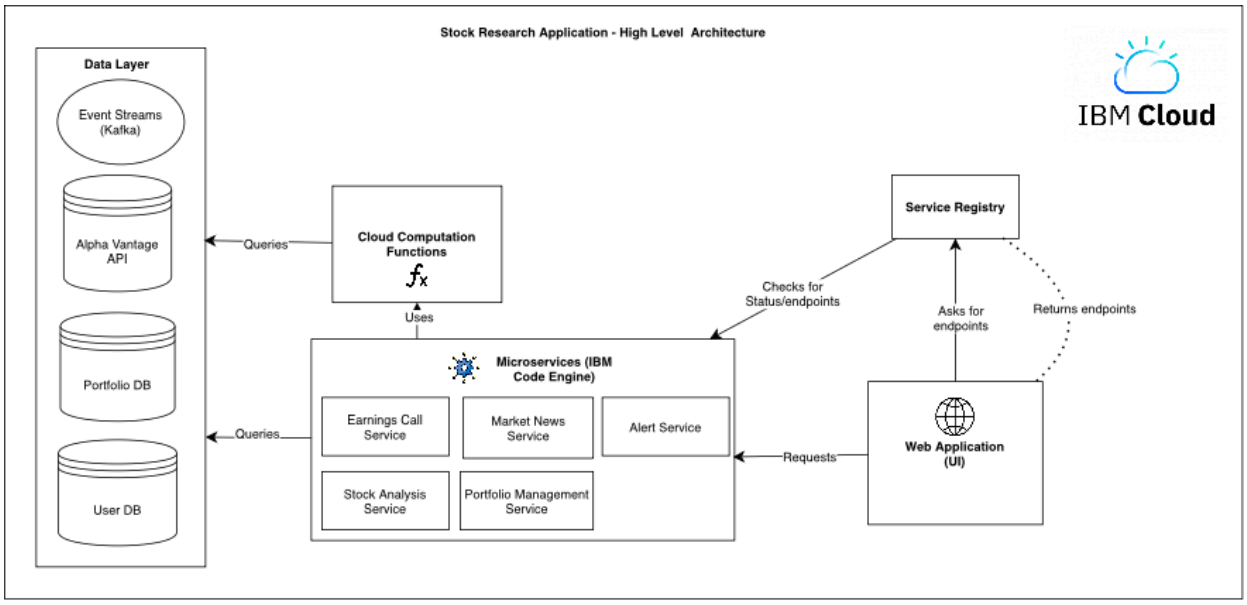
1. High-Level Architecture



The high-level architecture of the Stock Research Application consists of four primary runtime domains: the Web Application, the Microservices Layer, the Cloud Computation Functions, and the Data Layer, coordinated through a Service Registry. The architecture follows a modular microservice pattern deployed on IBM Cloud using Code Engine, Event Streams, and managed databases.

1.1. Web Application (UI)

The Web Application represents the user-facing front end of the system. It handles user interactions, sends requests to backend microservices, and renders the results returned from the system. The UI uses HTTP/HTTPS REST requests to communicate with backend services, and relies on the Service Registry to dynamically discover the URLs of these services.

Responsibilities:

- Render stock analysis results, portfolio information, earnings data, market news, and alert configurations
- Capture user inputs and forward them to backend services
- Display the results of analytics and data retrieval
- Remain decoupled from backend endpoint details through service discovery

---

1.2. Service Registry

The Service Registry provides dynamic service discovery, ensuring that the Web Application and other components can locate healthy, active instances of each microservice. This prevents the use of hardcoded URLs and enables resilience in the presence of autoscaling or redeployment.

Functions:

- Stores live service endpoints and health status
- Responds to endpoint discovery requests from the Web Application
- Receives registration and heartbeat updates from microservices
- Ensures routing remains consistent even as Code Engine scales services up or down

Operational flow:

1. Microservices register themselves on startup.
2. The Web Application requests the location of a specific service.
3. The registry returns the current valid endpoint.

This mechanism supports scalability, modifiability, and fault tolerance.

1.3. Microservices Layer (IBM Code Engine)

The microservices implement the business logic of the system and correspond to major application features. Each microservice is deployed independently as an IBM Code Engine application, enabling isolation, autoscaling, and independent deployment cycles.

Microservices include:

- Stock Analysis Service
- Market News Service
- Portfolio Management Service
- Earnings Call Service
- Alert Service

Responsibilities:

- Implement business workflows
- Issue calls to Cloud Computation Functions for analytics
- Retrieve and persist data in the Data Layer
- Publish or consume events from Event Streams where necessary
- Register and maintain their presence with the Service Registry

Communication:

- Exposes REST endpoints for UI and inter-service communication
- Invokes computation modules internally or through serverless functions
- Queries data sources to fulfill requests

1.4. Cloud Computation Functions (Analytics Layer)

Cloud Computation Functions represent the analytical layer of the system. These functions may run as IBM Cloud Functions (serverless) or as computation modules within microservice containers. They execute the analytical operations necessary for stock analysis, sentiment scoring, ranking, and portfolio computations.

Example computations:

- Technical indicators (moving averages, RSI, MACD, volatility metrics)
- Natural language sentiment scoring for news
- Ranking and searching earnings call transcripts
- Portfolio risk, return, and performance metrics

Characteristics:

- Stateless analytical tasks
- Scales on demand when deployed as serverless functions
- Interfaces directly with the Data Layer to retrieve raw data

---

1.5. Data Layer

The Data Layer provides persistent storage and integration with external financial data sources. It includes databases, external APIs, and event streaming infrastructure.

Components:

- User Database: Stores user accounts, credentials, and preferences
- Portfolio Database: Stores portfolio holdings, historical transactions, and computed metrics
- Alpha Vantage API: External market data provider for real-time and historical pricing
- Event Streams (Kafka): Used for asynchronous messaging, alert publication, and potential ingestion of live news feeds

Communication patterns:

- Microservices and computation functions query the databases via standard database connectors
- Computation functions retrieve raw market data through external API calls
- Event Streams act as publish-subscribe channels for asynchronous workflows

---

1.6. End-to-End Flow Summary

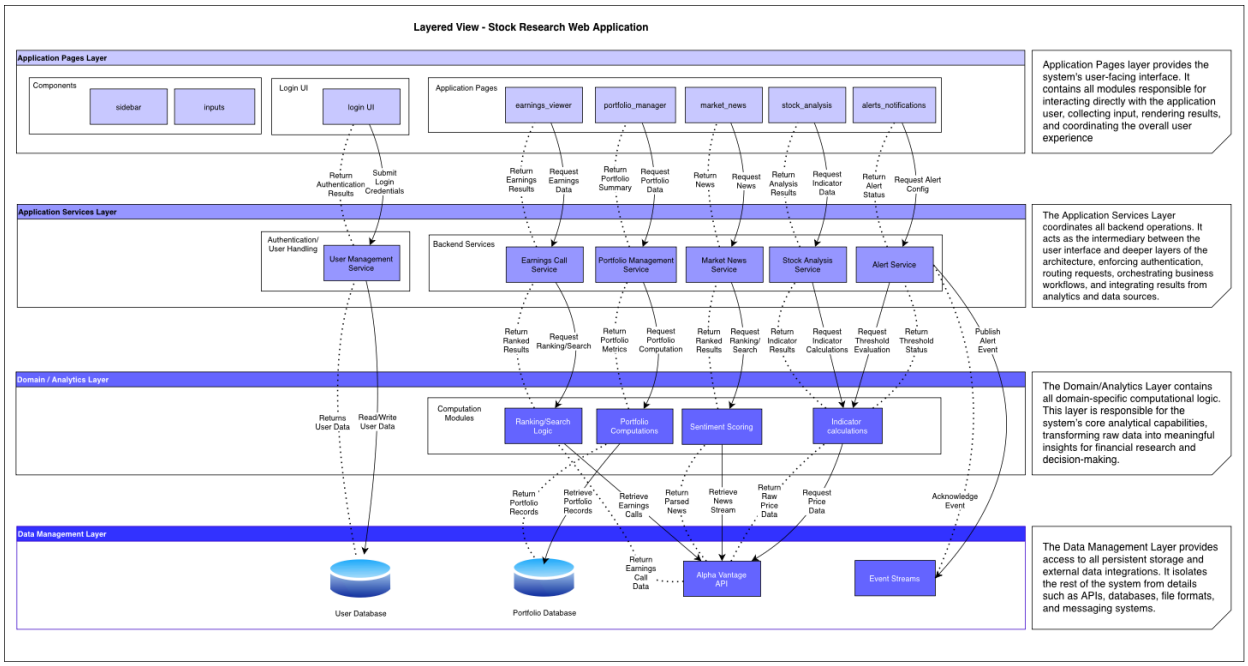
1. The user interacts with the Web Application UI.
2. The UI queries the Service Registry to determine the correct microservice endpoint.
3. The UI then sends an HTTP request directly to the corresponding microservice.
4. The microservice performs its logic, potentially involving:
    - Querying the Data Layer
    - Invoking Cloud Computation Functions
    - Publishing or consuming event streams
5. The microservice returns the processed result back to the UI.
6. The UI updates the display with the resulting data or analysis.

This structure ensures system scalability, resilience, and ease of modification.

2. Module Views

2.1. Module View – Layered

(Module Structure View Documentation — Following BCK Template)



2.1.1. Primary Presentation

The Layered View presents the structure of the Stock Research Web Application as a hierarchy of four architectural layers. Each layer groups modules with similar responsibilities and restricts dependencies to flow only downward. This separation supports modifiability, simplicity, and clear responsibility boundaries.

The four layers are:

1. **Application Pages Layer**
2. **Application Services Layer**
3. **Domain / Analytics Layer**
4. **Data Management Layer**

The accompanying diagram visualizes the modules within each layer, along with the allowed dependency relationships between them. Upward dotted arrows represent returned data, while downward solid arrows represent service requests or module dependencies.

2.1.2. Element Catalog

This section describes the elements (modules) in each layer and their roles.

Application Pages Layer (UI Layer)

Responsibilities:

- Provide all user-facing functionality
- Collect input (tickers, thresholds, credentials)
- Trigger backend operations
- Render the results of analytics and data queries
- Contain reusable UI components, login interface, and feature pages

Modules:

- **Components:** sidebar, inputs
- **Login UI:** user authentication entry point
- **Application Pages:**
    - earnings_viewer
    - portfolio_manager
    - market_news
    - stock_analysis
    - alerts_notifications

Rationale:
This layer isolates presentation concerns from business logic, supporting ease of UI modification without altering backend logic.

Application Services Layer (Backend Orchestration)

Responsibilities:

- Coordinate all backend operations
- Enforce authentication and user access control
- Route feature-specific requests to analytics and data layers
- Aggregate results into UI-ready outputs
- Provide logical “service endpoints” for each system capability

Modules:

- **User Management Service** (authentication, user profiles)

- **Stock Analysis Service**
- **Portfolio Management Service**
- **Market News Service**
- **Earnings Call Service**
- **Alert Service**

Rationale:
This layer provides a cohesive set of backend operations, enabling clean separation between UI and analytical/persistence concerns. It ensures business workflows remain consistent regardless of UI implementation.

Domain / Analytics Layer (Computation Layer)

Responsibilities:

- Perform the system's core analytical computations
- Compute stock indicators and financial metrics
- Analyze sentiment of market news
- Rank and filter earnings calls
- Compute portfolio performance summaries
- Transform raw data into structured insights

Modules:

- **Indicator Calculations**
- **Sentiment Scoring**
- **Ranking/Search Logic**
- **Portfolio Computations**

Rationale:
By centralizing computation, this layer ensures analytical logic is reusable across services and modifiable without affecting UI or data storage logic.

Data Management Layer (Persistence & External Data Integration)

Responsibilities:

- Manage storage and retrieval of all persistent system data
- Interface with the User Database and Portfolio Database
- Retrieve financial data from external APIs (e.g., Alpha Vantage)

- Handle event stream interactions for news and alerts
- Abstract database queries, API calls, and data-fetch operations

Modules / Resources:

- **User Database**
- **Portfolio Database**
- **Alpha Vantage API**
- **Event Streams**

Rationale:
This layer encapsulates all persistence and external data concerns, ensuring that higher-level modules depend on stable, abstracted data interfaces.

2.1.3. Context

The layered structure fits within the broader architecture of a microservice-based stock research platform. Each layer refines system responsibilities and restricts how modules interact. The user interacts only with the Application Pages Layer, which depends on the Services Layer. Analytical computations occur only in the Domain/Analytics Layer, and all persistent/external data is accessed exclusively through the Data Management Layer.

This structure prevents cross-layer interference and enforces a disciplined architectural approach.

2.1.4. Variability Guide

The Layered View supports several points of variation:

- **UI Variability:**
  Application Pages can be restyled or reorganized without affecting backend logic.
- **Service-Level Variation:**
  Backend services can be extended with new features (e.g., additional alert types) without modifying UI or analytics layers.
- **Analytics Extension:**
  New indicators, sentiment models, or ranking algorithms can be added within the Domain/Analytics Layer.

- **Data Source Variation:**
  The Data Management Layer supports replacing or expanding data sources (e.g., switching APIs or migrating databases) without requiring UI or service changes.

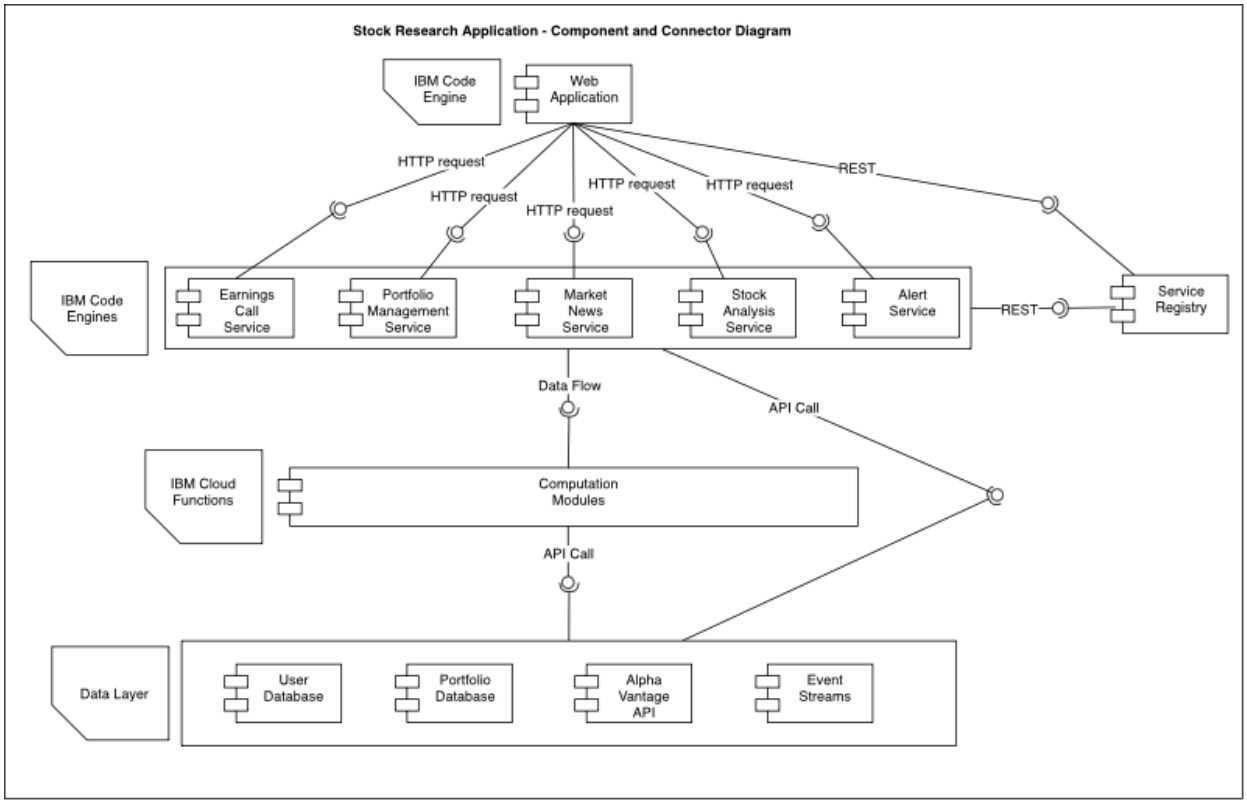
These variations highlight the architectural flexibility enabled by the layered approach.

2.1.5. Rationale

This architecture adopts a layered style to ensure:

- **High Modifiability:**
  Different concerns (UI, services, computation, persistence) are cleanly separated, reducing the impact of changes.
- **Reusability of Business Logic:**
  Analytical computations are isolated so multiple services can reuse them.
- **Clear Dependency Direction:**
  Dependencies flow downward only. No layer depends on a layer above it.
- **Scalability and Maintainability:**
  Each layer can evolve independently, allowing backend services and data sources to scale or change without redesigning the entire system.
- **Alignment with Architectural Requirements:**
  Supports authentication boundaries, scalable analytics, reliable data access, and extension of service features.

2.2. Component-and-Connector View



Stock Research Application — High-Level Runtime Architecture

The Component-and-Connector (C&C) diagram illustrates the runtime structure of the Stock Research Application. It identifies the system's major components—web application, backend microservices, computation modules, data sources, and the service registry—and shows how they interact through well-defined communication connectors. This view emphasizes communication pathways, runtime dependencies, and integration points across the architecture.

2.2.1. Web Application (UI)

The Web Application, deployed via IBM Code Engine, is the primary entry point for user interaction. It sends HTTP/HTTPS REST requests to backend services in order to retrieve stock data, portfolio results, news analyses, earnings information, and alert statuses.

In runtime:

- The UI queries microservices directly using REST requests.
- It consults the Service Registry to obtain the correct service endpoints, avoiding hardcoded URLs.

- It communicates outward only through standard HTTP connectors.

This ensures the UI remains decoupled from service deployment details and adapts automatically to scaling or redeployment of services.

2.2.2. Service Registry

The Service Registry supports dynamic service discovery.
It enables the web application and backend services to determine where microservices are currently deployed and whether they are healthy.

At runtime:

- Microservices register their endpoints with the registry.
- The Web Application requests “Where is this service located?”.
- The registry returns the current, valid endpoint.

Connectors between microservices and the registry are REST-based and used for registration, heartbeat updates, and lookup.

2.2.3. Microservices (IBM Code Engine)

The middle tier consists of a suite of independently deployed **microservices**, each hosted on IBM Code Engine. These services encapsulate the application's functional capabilities:

- **Earnings Call Service**
- **Portfolio Management Service**
- **Market News Service**
- **Stock Analysis Service**
- **Alert Service**

Each microservice:

- Exposes REST endpoints to receive requests from the UI or other services.
- Makes calls to computation modules to perform analytical processing.
- Retrieves or stores data through connectors to the Data Layer.
- Registers itself with the Service Registry.

The connectors used here include:

- REST connectors (UI to service, service to registry)
- Internal service invocation (services to computation modules)
- Database query connectors (services to databases)

This structure supports independent scaling, loose coupling, and fault isolation.

2.2.4. Cloud Computation Functions (Analytics Layer)

The Computation Modules, potentially deployed as IBM Cloud Functions, are responsible for all domain-specific computational work. These modules implement calculations such as:

- Technical indicator calculations
- Sentiment analysis
- Ranking and filtering of earnings calls
- Portfolio performance computations

Services invoke these modules through API Call connectors, enabling separation of business logic from presentation and data concerns. The computation modules also interact with external data sources such as the Alpha Vantage API.

Connectors include:

- API calls from microservices
- Data flow connectors to databases and external APIs

This design enables computation logic to scale independently and supports high-performance analytic operations.

2.2.5. Data Layer

The **Data Layer** encompasses persistent storage and external data sources:

- **User Database**
- **Portfolio Database**
- **Alpha Vantage API**
- **Event Streams (Kafka)**

Microservices and computation modules interact with these components using **query connectors** and **API calls**. Databases provide persistent storage for user and portfolio information. External APIs provide stock market data. Event Streams support asynchronous workflows, such as alert triggers or real-time news ingestion.

The connectors here represent:

- Database queries (SQL/NoSQL)
- External API requests
- Asynchronous messaging interactions

2.2.6. Runtime Interaction Summary

The overall runtime sequence is as follows:

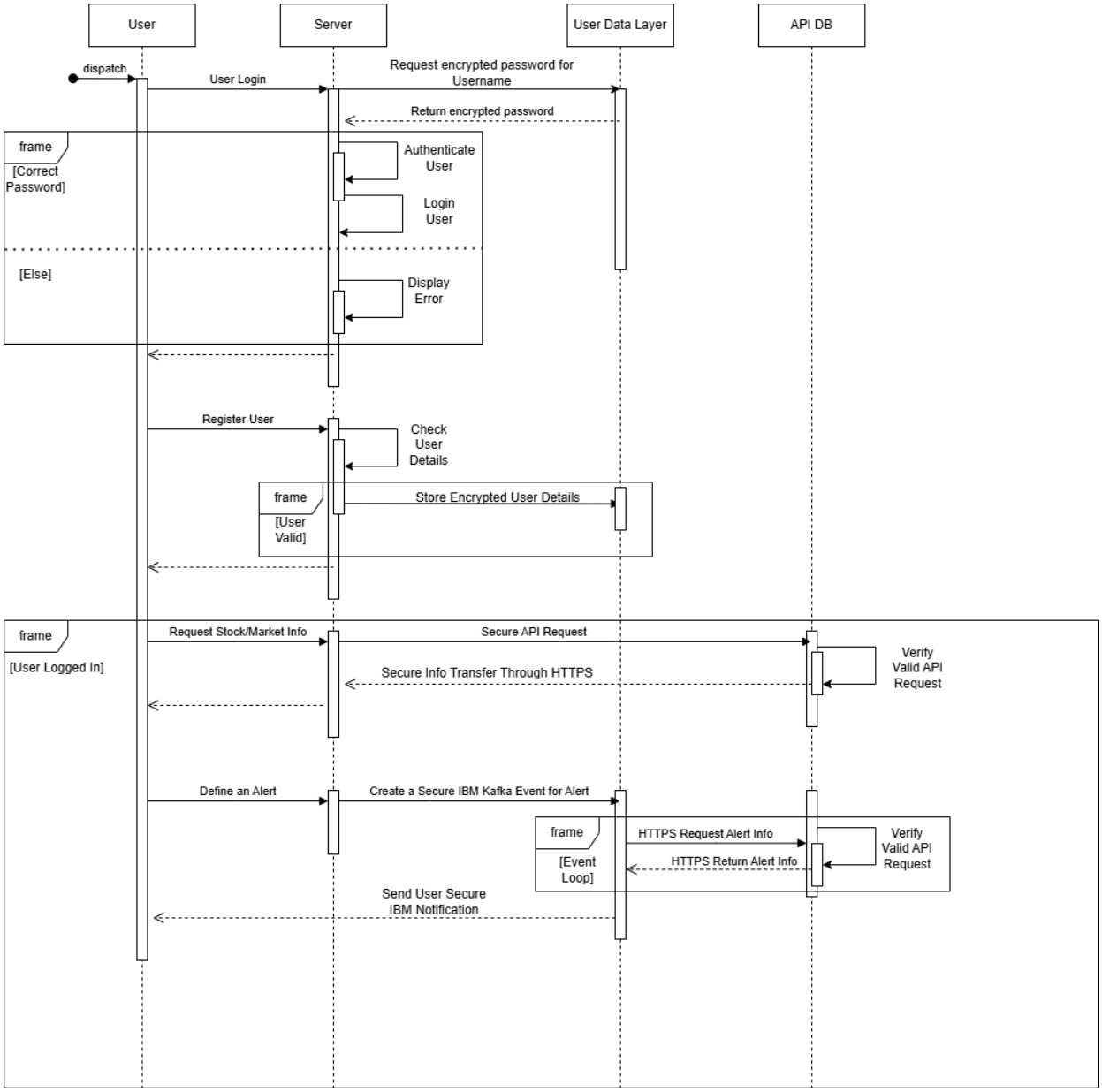
1. The user interacts with the Web Application.
2. The Web Application queries the Service Registry for current service endpoints.
3. The Web Application sends REST requests to the appropriate microservices.
4. Microservices perform their logic:
    - Querying databases
    - Invoking computation modules
    - Calling external APIs
    - Publishing or consuming event messages
5. Microservices return processed results back to the Web Application.
6. The UI updates the user interface with the returned data.

3. Quality Views

This section describes the system's quality views, to help understand the behaviors of the system:

(1) Security View using a sequence diagram to describe all measures taken to provide security.

3.1. Security View



Notes: All requests/returns from the server and databases are all secure through IBM's security or security from HTTPS.

Treat User Data Layer as a clump of all other IBM services being used other than the main code engine which is represented by Server. Other IBM services include our databases, event handlers, and notifications.

3.1.1. Purpose

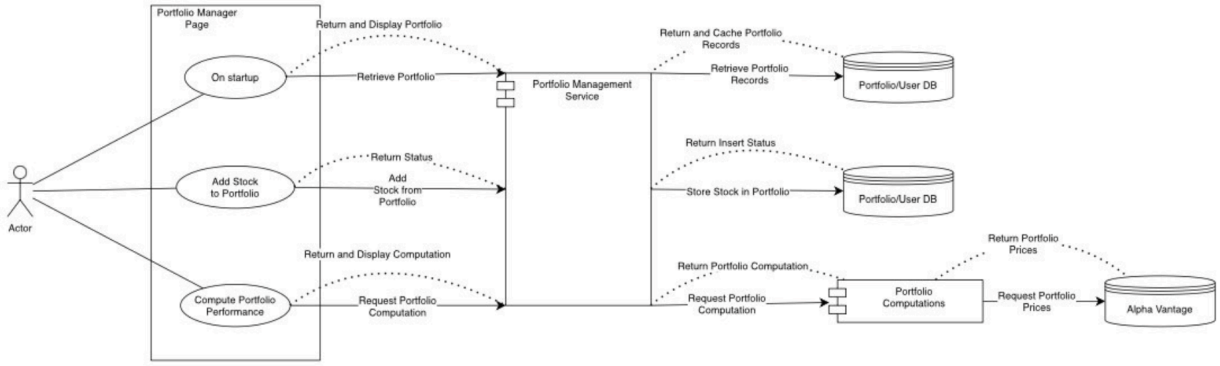
The purpose of the security view is to show how security interacts with the system and where it is present. In the sequence diagram we have identified all components that use some sort of security and show the links between these components. Most of our security is based on keeping user data and data transfers secure.

3.1.2. Security

These are the forms of security present in our system and shown in the sequence diagram:

- Validating passwords, user details, and API keys are all part of ensuring that data received is safe and applicable to the system.
- Secure Transfer Protocols:
    - HTTPS is a common protocol used to transfer information, but it is also known to be a safe transport protocol (The S at the end of HTTPS stands for Secure). Requests and returns to and from the API database all use HTTPS.
    - IBM Security Protocols: Our server, database, and many of our other services are all hosted on IBM cloud. They all use IBM security protocols for the safe transfer of data to and from these locations. IBM also uses HTTPS.

3.2. Communication View



3.2.1 Purpose

- Interaction across multiple microservices and external systems.
- Loose coupling between services
- Dynamic scaling and replacement of microservices
- Clear separation of responsibilities

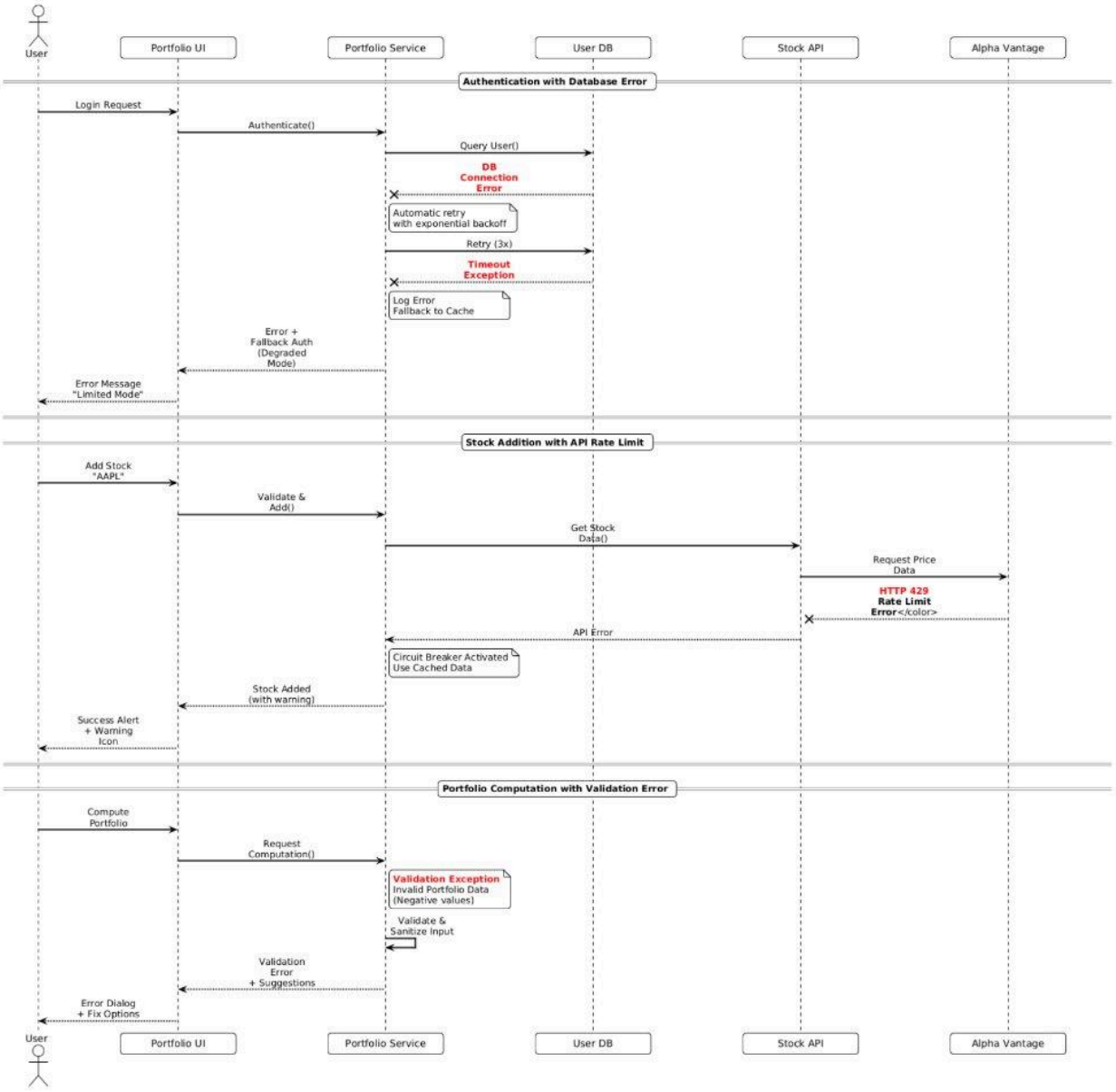
3.2.2 Implementation

- **Retry Mechanisms:** Automatic retry with exponential backoff for transient failures
- **Circuit Breaker Pattern:** Prevent repeated calls to failing services, enable auto-recovery
- **Fallback Strategies:** Cache-based degraded operation when real-time services unavailable
- **Rate Limit Management:** Intelligent request queuing for external API constraints
- **Comprehensive Logging:** Correlation IDs for distributed transaction debugging
- **Input Validation:** Sanitization and validation before processing to prevent downstream errors
- **User-Centric Error Handling:** Actionable error messages with clear recovery paths

Key Benefits

- Maintains 99.9% availability through graceful degradation
- Reduces MTTR (Mean Time To Recovery) via automated recovery mechanisms
- Ensures business continuity during partial system failures

3.3. Error/Exception Handling View



3.3.1 Purpose

• Ensure System Resilience: Maintain service availability during component failures and network issues
• Prevent Cascade Failures: Isolate faults to prevent system-wide outages
• Preserve Data Integrity: Guarantee consistency despite transient failures
• Enhance User Experience: Provide meaningful feedback and graceful degradation

3.3.2 Implementation

- **Retry Mechanisms:** Automatic retry with exponential backoff for transient failures
- **Circuit Breaker Pattern:** Prevent repeated calls to failing services, enable auto-recovery
- **Fallback Strategies:** Cache-based degraded operation when real-time services unavailable
- **Rate Limit Management:** Intelligent request queuing for external API constraints
- **Comprehensive Logging:** Correlation IDs for distributed transaction debugging
- **Input Validation:** Sanitization and validation before processing to prevent downstream errors
- **User-Centric Error Handling:** Actionable error messages with clear recovery paths

Key Benefits

- Maintains 99.9% availability through graceful degradation
- Reduces MTTR (Mean Time To Recovery) via automated recovery mechanisms
- Ensures business continuity during partial system failures