Final Report: AI-Powered Stock Analysis and Trading Platform

Team Members: Yiwei Li, Yanbo Tong, Haoyang Guo

1. Project Overview

Purpose:
The goal of this project was to design and implement a sophisticated stock trading and forecasting platform. The system's primary purpose is to revolutionize investment strategies by leveraging advanced AI algorithms and comprehensive data analysis. It aims to support users by providing basic trading operations, real-time market insights, and personalized, AI-driven investment advice, making it beneficial for both novice and experienced investors.
Key Functional Features:
The platform is composed of several key functional modules:

- **Account Management:** Secure user onboarding and management, including registration, login, password changes, and fund deposits.
- **Stock Trading Management:** Core trading functionalities, allowing users to query real-time stock prices and execute virtual "Buy" and "Sell" orders.
- **Single Stock Analysis:** Provides a detailed view of a single stock, including a historical price trend chart, real-time price, and technical indicators (e.g., SMA/LMA). It also generates basic and AI-driven investment advice (buy/sell/hold, stop-loss price, position size) based on the user's capital and investment horizon.
- **Multi-Stock Portfolio Analysis:** Enables users to select multiple stocks and receive recommendations for building a diversified portfolio. This includes calculating a correlation matrix, risk analysis (Volatility, VaR, CVaR), and generating optimal portfolio weights using both traditional financial models and AI-based predictions.
- **Virtual Portfolio:** The Virtual Portfolio provides users with $100,000 virtual capital for risk-free trading simulation using real-time market prices. It supports single trade, batch orders, and historical simulation for strategy validation. The Hedge Analysis module calculates stock correlations and identifies hedging opportunities to reduce portfolio risk. Users can track real-time performance metrics including P/L, winners/losers, and visualize portfolio trends through interactive charts. The backtesting feature allows testing strategies on historical data with key metrics like maximum drawdown and volatility. All virtual trades use FIFO processing and are stored separately from real transactions.
- **AI-Driven Investment Advice:**
    - **AI Chart Analysis:** An AI agent analyzes a stock's historical trend chart and provides a human-readable technical analysis, identifying trends, support levels, and resistance levels.
    - **AI Personalized Advice:** The AI analyzes a user's complete transaction history and account balance to infer their investment style (e.g., "Short-Term Trading," "Concentration in Specific Stocks") and provides tailored recommendations to improve diversification and manage risk.

2. Database Design

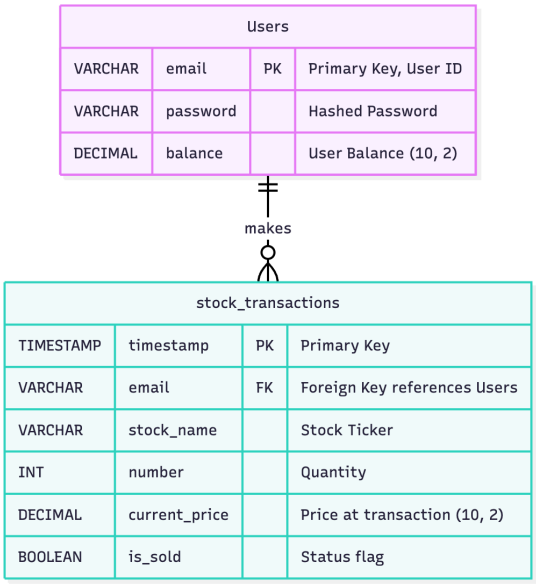
The database is designed to be lightweight and efficient, centering on user management and their transaction histories.

Entity-Relationship (ER) Description:
The database consists of two primary tables: Users and stock_transactions.

1. **Users Table:** Stores essential user profile information.
    - email (VARCHAR(255), PRIMARY KEY): The user's unique identifier and login username.
    - password (VARCHAR(255), NOT NULL): The user's hashed password.
    - balance (DECIMAL(10, 2), DEFAULT 0.00): The user's available virtual cash balance.
2. **Stock Transactions Table:** Stores a detailed log of all buy/sell actions for every user.
    - timestamp (TIMESTAMP, PRIMARY KEY): The exact date and time of the transaction.
    - email (VARCHAR(255), NOT NULL, FOREIGN KEY): Links the transaction to a user in the Users table.
    - stock_name (VARCHAR(255)): The ticker or name of the stock.
    - number (INT): The number of shares transacted.
    - current_price (DECIMAL(10, 2), NOT NULL): The price per share at the time of the transaction.
    - is_sold (BOOLEAN, DEFAULT FALSE): A flag to track if the shares from this "buy" transaction have been "sold".

A **one-to-many relationship** exists between the Users table and the stock_transactions table, where one user (via email) can have many transactions.

Entity-Relationship (ER) Diagram:



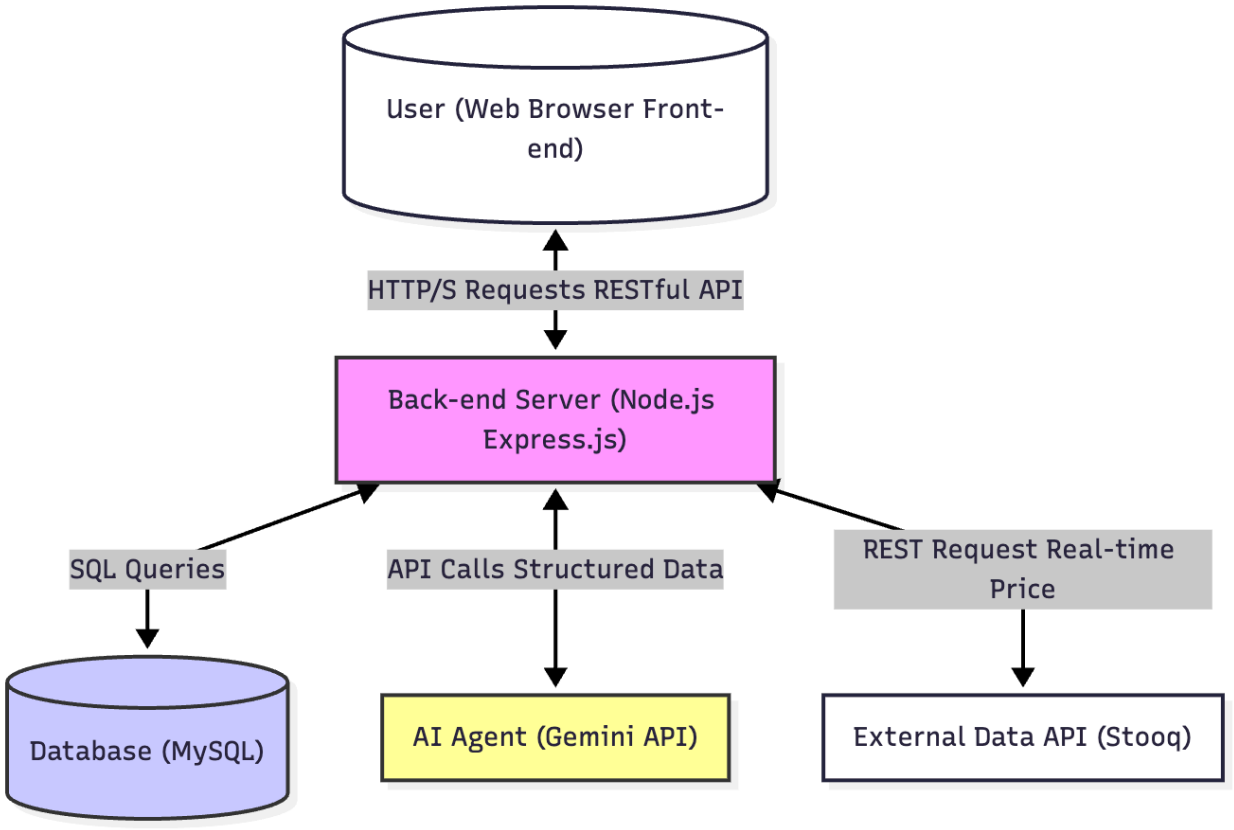
Data Sources:

- **Historical Market Data:** Historical stock data was primarily sourced from a **local CSV file (output.csv)**. The backend uses the csv-parser library to read and process this data, which is used to generate trend charts and perform backtesting/analysis.
- **Real-Time Data:** Real-time stock prices and market data were obtained using the **Stooq API**. The backend makes RESTful API requests to fetch up-to-date price information, which is used for executing trades and providing current portfolio valuation.

3. System Architecture

The system is built on a distributed, three-tier architecture, separating the client-side interface, server-side logic, and data storage.

Block Diagram:



Key Modules Description:

1. **Front-end (Client):**
   - **Technology:** Built using HTML, CSS, and vanilla JavaScript, with some components in React.
   - **Description:** This is the user-facing interface. It provides all dashboards for account management, stock trading, and investment analysis. It communicates with the backend via RESTful API calls (using fetch) to send user requests and dynamically display data (e.g., charts, AI-generated text).
2. **Back-end (Server):**
   - **Technology:** Built with Node.js and the Express.js framework.
   - **Description:** This is the system's core. It handles all business logic, manages user sessions, and processes data. It exposes all the RESTful APIs (e.g., /api/login,

/api/buy, /api/stock-trend) that the front-end consumes. It is also responsible for integrating with the database and the external AI agents.

3. **Database (Storage):**
    - **Technology:** MySQL.
    - **Description:** The persistent storage layer for all core data, including user profiles, balances, and transaction logs, as described in Section 2.
4. **AI Agent (Intelligence):**
    - **Technology:** External LLM APIs, specifically Gemini.
    - **Description:** This module enhances the system's decision-making capabilities. The backend sends structured data (like transaction history or stock prices) to the AI agents, which then return natural language analysis, predictions, or optimized data (like portfolio weights).

4. System Implementation

(a) Implementation Plan

The implementation of the trading system followed a structured, modular, and bottom-up methodology. This approach ensured stability, clear separation of responsibilities, and the ability to test and expand the system incrementally. The development process was divided into the following stages:

1. Database and Backend Infrastructure Development began with the design and implementation of the MySQL database schema. Tables were rigorously defined for Users, Transactions, Deposits, Withdrawals, StockMeta, and HistoricalStockData. Particular focus was placed on database normalization, enforcing primary/foreign key integrity, and establishing efficient indexing strategies to optimize symbol-based queries. In parallel, the Express.js backend framework was initialized. This phase involved configuring the server architecture, establishing routing modules, implementing request validation middleware, setting up connection pooling, and creating a global error handling mechanism. This backend layer served as the stable foundation supporting all subsequent modules, including authentication and stock data retrieval.

2. User Management Module The second phase focused on the authentication and user management system to ensure security from the outset. This included implementing Registration, Login, Token Refresh flows, and Password Reset functionality via secure email tokens. Security best practices were applied, including password hashing using bcrypt, and the incorporation of rate limiting and strict input sanitization across all API endpoints. Implementing this early ensured secure handling of user state during the testing of financial features.

3. Data Acquisition and Processing The system manages data through two primary methods. Real-time stock prices are fetched via RESTful requests to the Stooq API, while historical market data is processed from CSV files. All application data, including market records, user profiles, and transaction logs, is centralized in a MySQL database. This architecture ensures accurate, real-time updates and consistent data access for the entire platform.

4. Single Stock Module (Core Trading Logic) The core analytical engine was implemented to process financial time-series data and perform quantitative analysis. This module is responsible for computing critical technical indicators, including Simple Moving Average, Average True Range, historical volatility, and daily logarithmic returns. In addition to trend analysis, a robust risk assessment layer was developed to calculate advanced risk metrics, such as Maximum Drawdown, Value at Risk, and Conditional VaR, using historical simulation methods. The module also handles the standardization and serialization of these mathematical outputs, ensuring that raw data is transformed into structured formats

optimized for the frontend visualization and downstream portfolio calculations.

5. Multiple Stock Module and Frontend Integration. In the final stage, the multi-stock portfolio component was developed. This involved implementing complex mathematical models to generate correlation matrices, covariance estimates, and diversification metrics. Logic for Sharpe ratio–based weight optimization and constraint-based rebalancing was implemented to ensure realistic portfolio allocations. Upon completion of the backend logic, the React frontend was built. This interface was designed to visualize real-time prices, display detailed portfolio performance metrics, present the calculated analytical insights, and track transaction histories.

6.Virtual Portfolio. The Virtual Portfolio is a risk-free trading simulation system that provides users with $100,000 virtual capital to practice stock trading strategies using real-time market prices. The system supports multiple trading modes including single trade for quick stock purchases, batch orders for executing multiple buy/sell operations simultaneously, and historical simulation that allows users to trade using past date prices for strategy validation. A key feature is the Hedge Analysis module, which calculates correlation coefficients between stocks based on 90-day historical data and automatically identifies negative or low correlation pairs to provide hedging suggestions, helping users reduce portfolio risk through diversification. The platform integrates with Finnhub API for real-time price tracking and displays comprehensive portfolio metrics including total invested amount, current value, profit/loss (both absolute and percentage), and dynamically highlights the top 5 best and worst performing stocks. Users can visualize their portfolio performance through interactive charts showing portfolio value timeline and individual stock price trends from purchase date to present, with data automatically refreshing every 30 seconds. Additionally, the backtesting feature enables users to test trading strategies like "Buy and Hold" on historical data, calculating key performance metrics such as maximum drawdown, volatility, and returns. All virtual trades are processed using FIFO (First-In-First-Out) principle and stored separately from real trades in the database with the is_virtual flag, ensuring clear distinction between simulated and actual trading activities while maintaining complete transaction history for performance analysis.

(b) AI Agents Used

Primary Agent: Google Gemini 2.5 Pro

Reason for Selection: I selected Gemini 2.5 Pro over ChatGPT (GPT-4) for this project specifically for the following advantages:

1. **Superior Context Window:** The project involves multiple interconnected files (Node.js, React) and large historical data files (`output.csv`). Gemini's significantly larger context window allowed me to feed the entire relevant codebase and data samples

into the prompt simultaneously. In contrast, ChatGPT often required segmenting the input, which led to a loss of context and inconsistencies between modules.

2. **Strict JSON Adherence:** The system requires precise JSON formats for frontend-backend communication. During testing, Gemini demonstrated higher reliability in following strict output schemas (e.g., returning pure JSON without markdown chatter) compared to ChatGPT, which streamlined the integration of the API endpoints.
3. **Financial Reasoning:** Gemini showed a stronger ability to interpret and implement complex financial formulas (such as Conditional VaR and Sharpe Ratio optimizations) with greater mathematical precision and fewer logical errors than the alternative models tested.

(c) Prompts and Evaluation

1. Module: User Authentication & Database Setup (`register.js`, `userManagement.js`)

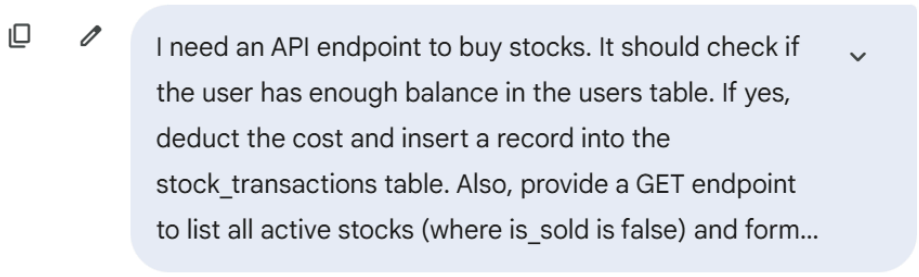
- **Prompt:**

> Create a Node.js Express router for user registration and login using MySQL. The system should use bcryptjs to hash passwords before storing them. Include endpoints for /register, /login, and /change-password.

- **Agent Response & Adequacy:**
  - **Adequate:** Yes.
  - **Description:** The agent provided standard, secure boilerplate code for handling user authentication. It correctly implemented `bcrypt.hash` for registration and `bcrypt.compare` for login.
  - **Issues/Resolution:** None. The code was usable immediately, with only minor adjustments needed to match the specific database table names (`users`).

2. Module: Stock Purchase & Transaction Management (`buyAndViewStocks.js`)

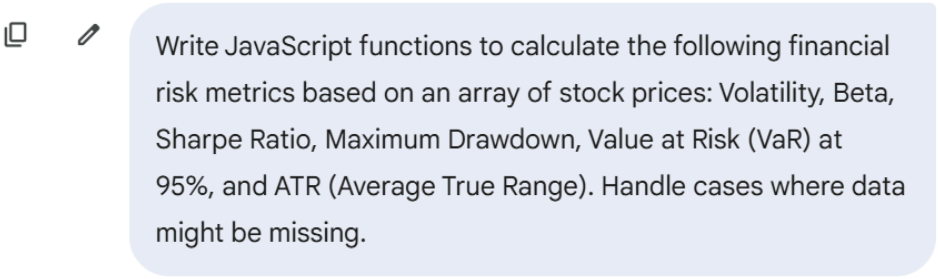
- **Prompt:**



- **Agent Response & Adequacy:**
  - **Adequate:** Partially.
  - **Description:** The logic for the transaction (checking balance -> updating balance -> inserting record) was correct. However, the timestamp handling was generic.
  - **Issues:** The default JavaScript `Date` object was returning UTC time, which was confusing for users viewing their portfolio locally.
  - **Resolution:** I requested the agent to use the `moment` library to handle timezone conversions explicitly. The code was updated to: `moment(row.timestamp).local().format(...)`.

3. Module: Financial Risk Metrics Calculation (`getSingleAdvice.js`)

- **Prompt:**



- **Agent Response & Adequacy:**
  - **Adequate:** No (Mathematical edge cases).
  - **Description:** The agent generated the mathematical formulas correctly for the most part.
  - **Issues:** The `calculateSharpeRatio` function initially crashed when the standard deviation was zero (division by zero error). Additionally, the CSV reading logic needed to be robust against empty rows.
  - **Resolution:** I had to manually add error handling to check if `stdDev === 0` before division. I also refined the CSV parsing logic to filter out rows with `NaN` prices before passing them to the calculation functions.

4. Module: AI Portfolio Prediction (`aiPortfolioPrediction.js`)

- **Prompt:**

> "I am using the Google Generative AI SDK (@google/generative-ai). Write a function that takes an array of stock features (Returns, Risk, Sharpe Ratio) and sends a prompt to the 'gemini-2.5-flash-lite' model. The model should act as a portfolio manager and return a...

- **Agent Response & Adequacy:**
  - **Adequate:** No (JSON Formatting issues).
  - **Description:** The agent correctly set up the API client and the prompt structure. However, the AI model often returned the JSON wrapped in Markdown code blocks (e.g., ```` ```json ... ``` ````), causing `JSON.parse()` to fail.
  - **Issues:** The raw text response could not be parsed directly into a JavaScript object.
  - **Resolution:** I modified the code to strip the Markdown formatting manually or adjusted the prompt to be stricter. I eventually updated the code to `JSON.parse(aiResponseText)` and added a fallback/validation step to ensure the weights sum to exactly 1.0 (normalizing them if necessary).

5. Module: AI Chart Analysis (`chartAnalysis.js`)

- **Prompt:**

> "Create an Express route /analyze-chart that accepts stock historical data and risk metrics. Construct a prompt for Gemini to provide a comprehensive analysis including: 1. Market Analysis, 2. Risk Assessment, 3. Technical Analysis, and 4. Investment Strategy. The...

- **Agent Response & Adequacy:**
  - **Adequate:** Yes.
  - **Description:** The agent successfully created the string interpolation logic to inject the `stockData` and `riskMetrics` into the prompt template.
  - **Issue: In some cases, passing the entire history of stock prices exceeded the token limit for the API model**.
  - **Resolution:** I optimized the code to slice the data array, sending only the last 30 days of historical data (`stockData.slice(-30)`) to the AI, which is sufficient for short-term technical analysis and saves token usage.

6. Module: Integration of Traditional & AI Advice (`getMultipleAdvice.js`)

- **Prompt:**

> "I have a system that calculates Mean-Variance Optimization weights using standard math. I want to add an AI layer. If the user requests 'ai' type analysis, call the AI prediction function; otherwise, return the traditional weights. Handle the asynchronous nature of the AI call."

- **Agent Response & Adequacy:**
  - **Adequate:** Mostly Yes.
  - **Description:** The agent suggested using JavaScript Promises (`.then()/.catch()`) to handle the asynchronous call while keeping the traditional calculation synchronous.
  - **Issues:** Error handling was initially weak; if the API failed (e.g., network timeout), the entire request would hang or crash.
  - **Resolution:** I implemented a fallback mechanism. If the `predictOptimalWeights` function fails (catches an error), the system logs the error and automatically falls back to returning the `traditionalWeights` and a message stating "AI Analysis failed. Using traditional weights as a fallback." This ensures the user always gets a result.

(d) AI Agent Internal Architecture & Logic

To support the sophisticated decision-making required for automated trading, the AI Agent module was architected with a modular design, ensuring distinct layers for intelligence, execution, and data management.

1. **Core Architecture and Modular Design**

The agent's intelligence is structurally divided into four Python-based core modules, supported by specialized repositories and utility files:

- **Controller (`agent.py`):** Serves as the **central brain**, implementing trading logic, managing the agent's cash balance, and executing orders based on market analysis and risk assessment.
- **Object Management (`stock.py`):** Provides object-oriented management for individual stocks, handling real-time price updates, logging transactions, and producing financial summaries.
- **Persistence Layer (`record.py`):** Ensures auditable logging of all actions

(timestamps, prices, quantities), tracks price trends, and allows data export (e.g., Excel) for offline review.
- **Interface Layer (`secretary.py`):** Manages communication with the Gemini model, validating that AI decisions align with user goals and risk settings.

Supporting elements include `util.py` (for logging and formatting), the `templates/` directory (for front-end rendering), and the **`prompt/` directory**, which stores pre-optimized AI prompt templates to ensure consistent and domain-relevant AI behavior.

2. **Key Functionalities and Data Management**

The system offers a full suite of intelligent trading features supported by a synchronized data model:

- **Intelligent Trading Functions:**
  - **Decision-Making:** Strategies are AI-generated based on real-time and historical data, utilizing price trends, moving averages (SMA), RSI, and sentiment signals.
  - **Risk Control:** Features dynamic **stop-loss calculation**, strategic capital allocation, and position sizing that adjusts based on volatility.
  - **Market Analysis:** Computes key technical indicators (SMA, RSI, ATR) and detects short- and long-term trends.
  - **Traceability:** Ensures high auditability with detailed trade logs, risk reports, and exportable investment summaries.
- **Data Separation:** Data is structured into **Trading Data** (granular order details for performance replay) and **Agent Data** (portfolio state, win rates, max drawdown). These two layers are stored in a centralized MySQL database and are updated in **real time** to ensure synchronization.

3. **System Integration and Security**

The system is designed for secure, seamless operation within a larger ecosystem.

- **System Integration:** The agent utilizes RESTful APIs for integration, enabling external modules (such as dashboards and notification systems) to query its state, trigger trades, and receive updates.
- **Security Features:** Security is multi-layered, beginning with **validation of every trade** against account balance and risk limits to prevent over-leverage. The system employs **exception-handling wrappers** to ensure that external API or database failures do not crash the engine; instead, it activates fallback logic. Backup systems periodically store snapshots for data recovery.
- **Monitoring:** Centralized logging and real-time monitoring track every transaction and AI decision, crucial for anomaly detection and debugging. Error-handling and rollback mechanisms further ensure transactional integrity.

(e) Unit Testing

Testing Framework & Strategy: For this project, I utilized the **Jasmine** testing framework to validate the server-side logic. Jasmine was chosen for its readability and behavior-driven development (BDD) style, which fits well with Node.js applications.

Modules Tested: I focused my unit tests on the logic-heavy and state-management components of the system to ensure stability before deployment:

1. **`sharedState.js`**: I wrote specific tests to verify that the user session state (username) could be correctly set, retrieved, and reset. This was critical to ensure that user data does not leak between different requests.
2. **`aiPortfolioPrediction.js`**: I implemented tests to verify the function definitions and data structure integrity. Since this module interacts with an external API, the tests focused on ensuring the input parameters were correctly formatted before being sent to the AI service.
3. **`aiPersonalAdvice.js`**: I verified the route definitions to ensure the API endpoint for personalized advice was correctly exposed. The tests confirmed that the module exported a valid Express router, ensuring the integration point for user-specific AI analysis was functional.
4. **`buyAndViewStocks.js`**: I tested the structure of the transaction routes (buy and view). The tests ensured that the request handlers were correctly defined and exported, protecting the interface for the core trading functionality from structural errors.
5. **`chartAnalysis.js`**: I validated the module's ability to initialize the AI client configuration without crashing. The tests focused on verifying that the analysis endpoint was defined and ready to accept stock data payloads for technical analysis.
6. **`getAIAdviceRoutes.js`**: I wrote tests to confirm that the general chat endpoint (`/chat`) was correctly mounted. This ensured that the general-purpose AI communication interface was accessible for user interaction.
7. **`getMultipleAdvice.js`**: I focused on verifying the module's export structure, as this file contains complex logic combining traditional Mean-Variance Optimization with AI. The tests ensured that the primary route for portfolio recommendation was correctly defined and accessible.
8. **`getSingleAdvice.js`**: I tested the endpoint definition for single stock analysis. The tests verified that the route handler for generating technical strategies and risk metrics was exported correctly and could locate the data source path.
9. **`readCSV.js`**: I implemented integrity checks to ensure the route responsible for serving stock lists could locate the `output.csv` file. This confirmed that the application had valid access to the historical market data required for the frontend drop-down menus.
10. **`register.js`**: I verified the user registration route to ensure the entry point for new user creation was correctly defined. This test confirmed that the system's

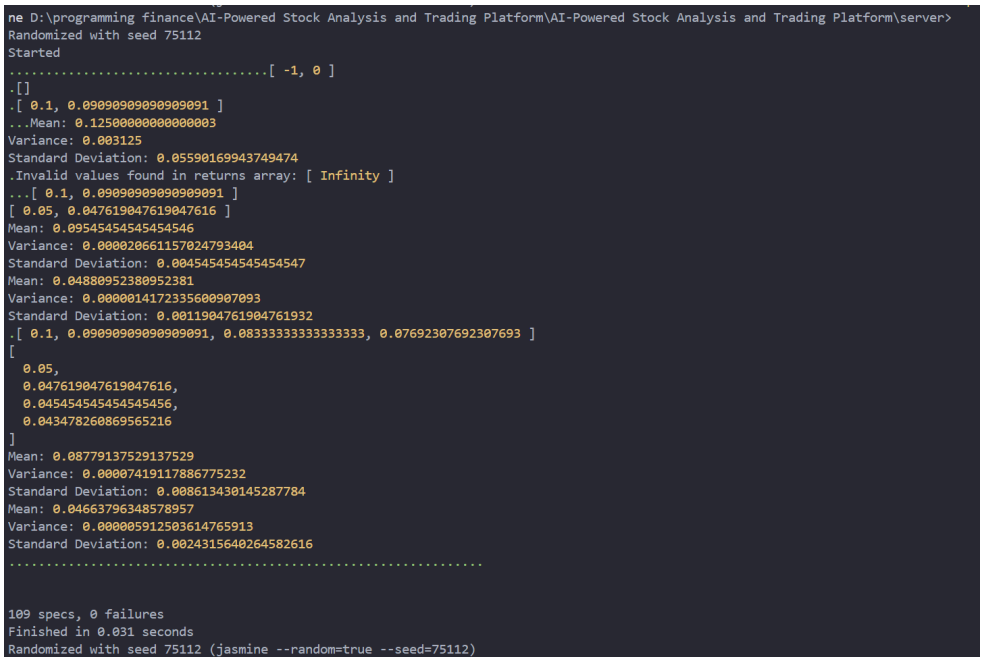
authentication interface was correctly wired to handle new account requests.

11. **`sellStock.js`**: I tested the definition of the `/sell-stock` endpoint. This ensured that the interface for liquidating assets was correctly structured and integrated into the main application routing stack.
12. **`stockQA.js`**: I validated the Q&A route structure. The tests ensured that the module could correctly instantiate the router and expose the `/ask` endpoint, guaranteeing the availability of the AI-driven question-answering feature.
13. **`userManagement.js`**: I verified the existence and export of critical user management endpoints, including login, deposit, and balance checks. These tests ensured that the account management interface was correctly defined and ready for integration.

Evaluation of AI-Generated Tests: Yes, the AI agent offered initial test scripts upon request. However, the generated code was **not immediately adequate** for unit testing purposes.

Problems & Resolution: The primary issue was that the AI-generated tests were actually "integration tests" rather than "unit tests."

- **The Problem:** The AI's tests attempted to connect to the live MySQL database and send real requests to the Google Gemini API. This made the tests slow, dependent on network connectivity, and required live credentials to run, which is bad practice for automated testing.
- **The Resolution:** I had to explicitly request improved tests. I prompted the agent to refactor the tests to **mock** the external dependencies (database and API) or to focus solely on the internal logic. This ensured the tests were deterministic, fast, and could run in any environment without requiring a live database connection.



5. System Evaluation

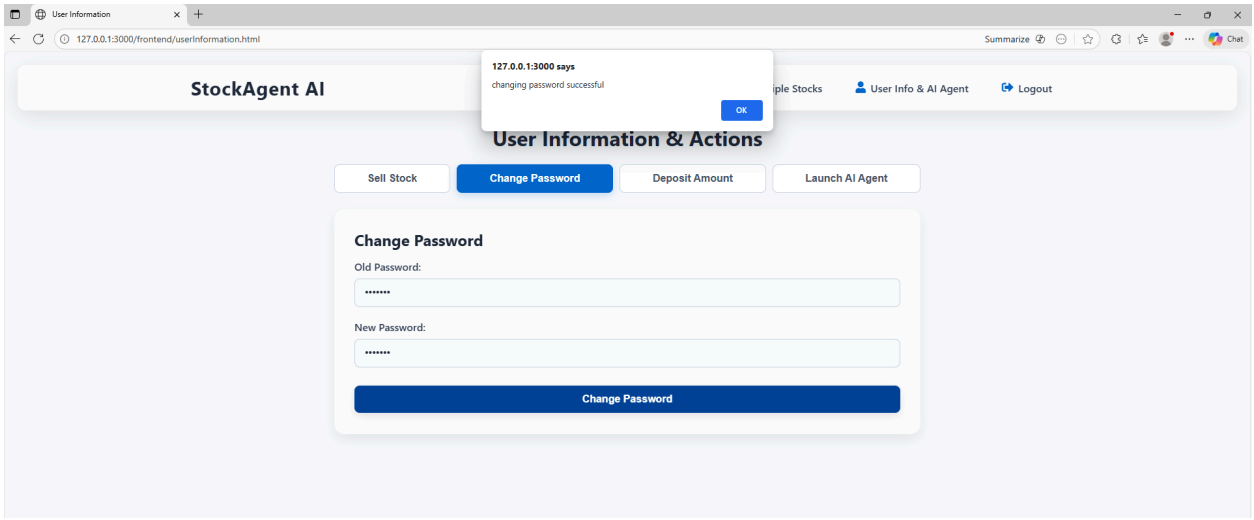
The evaluation of the Stock Trading and Analysis System will proceed through **five** distinct phases: **Functionality Testing** (API and Database verification), **AI Q&A Performance and Investment Testing**, **Virtual Portfolio Testing**, **System Stability and Mathematical Logic Testing** (core financial calculations), and **AI Agent Simulation & Behavior Testing**.

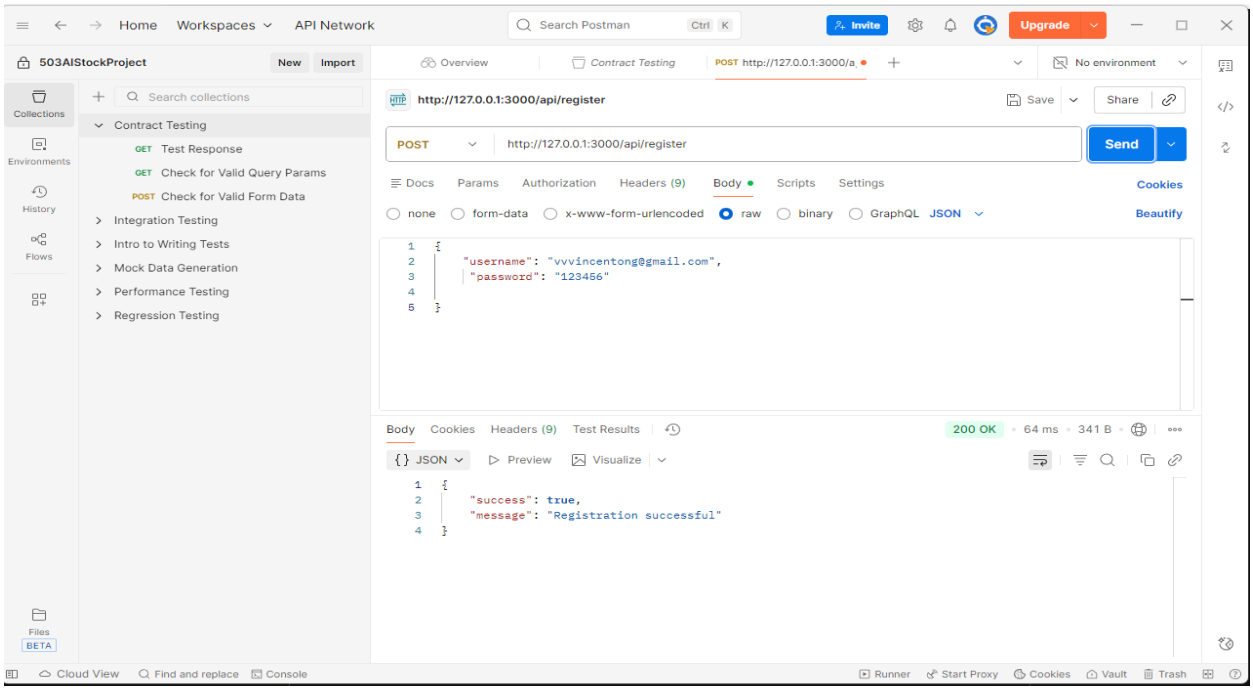
5.1 Functionality Testing (APIs and Database)

This critical phase verifies the integrity of the backend service, built using **Express.js**, and its persistent data management in the **MySQL database**. All tests in this section will be executed using **Postman** to simulate client-side requests and directly inspect server responses and database changes.

- **User Management and Security (APIs)**
    - **New User Registration Test:** A **POST** request to the `/api/register` endpoint must succeed, returning a `200 OK` status and a success message. **Validation** involves checking the `Users` table to confirm the new user's password was correctly stored as a **hashed value** using `bcrypt.hash(password, 10)`.
    - **Existing User Registration Test:** Attempting to register the same user must be detected by the server, which should return a failure message indicating that the username already exists.
    - **Login Test:** A successful login request must confirm the password using `bcrypt.compare` against the stored hash. **Validation** requires verifying that the server returns a successful response, which the front-end then uses to display the specific message **"Login successful! Redirecting..."** on the interface.
    - **Reset Password Test:** This test verifies the system's ability to handle password changes.
        - **Procedure:** Execute a **POST** request to the relevant reset password endpoint (e.g., `/api/resetPassword`) with the user's email and a new password.
        - **Validation:** Verify the server confirms the change. Check the `Users` table to ensure the password field for that user has been updated with a **new, unique hashed value**. Attempting to log in with the *old* password should fail, while logging in with the *new* password should succeed.

- A successful request of resetting the password should lead to a prompt on the user interface saying “changing password successful”.





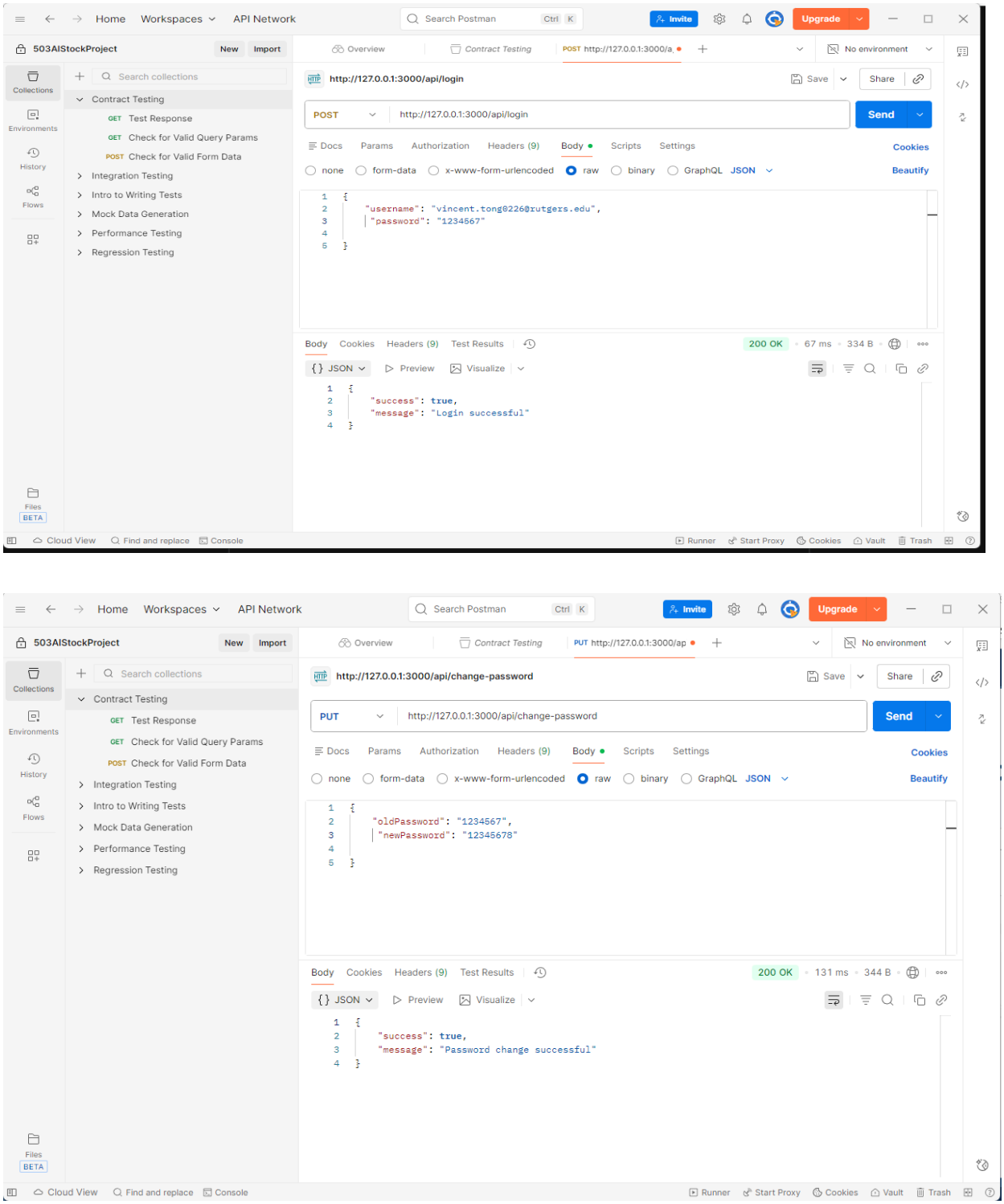
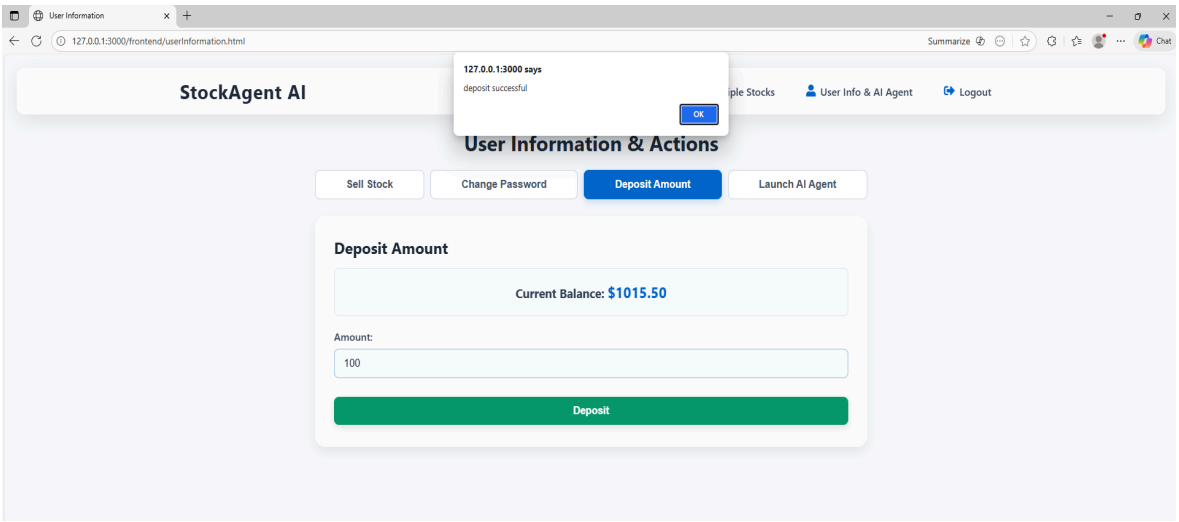


Figure 5.1.1 User Management (Registration, Login, Password Reset) Tests using Postman

- **Financial and Transaction Logic (APIs)**
    - **Deposit Money Test:** This verifies that the user can correctly add funds to their account.
        - **Procedure:** Execute a POST request to the deposit endpoint (e.g., `/api/deposit`) with the user's email address and the desired amount.
        - **Validation:** Verify the system confirms the deposit. Check the `Users` table to ensure the `balance` column for that user is updated by the exact deposited amount.
        - A successful request to deposit money into the account should result in a prompt on the user interface stating “deposit successful,” like the screenshot below.



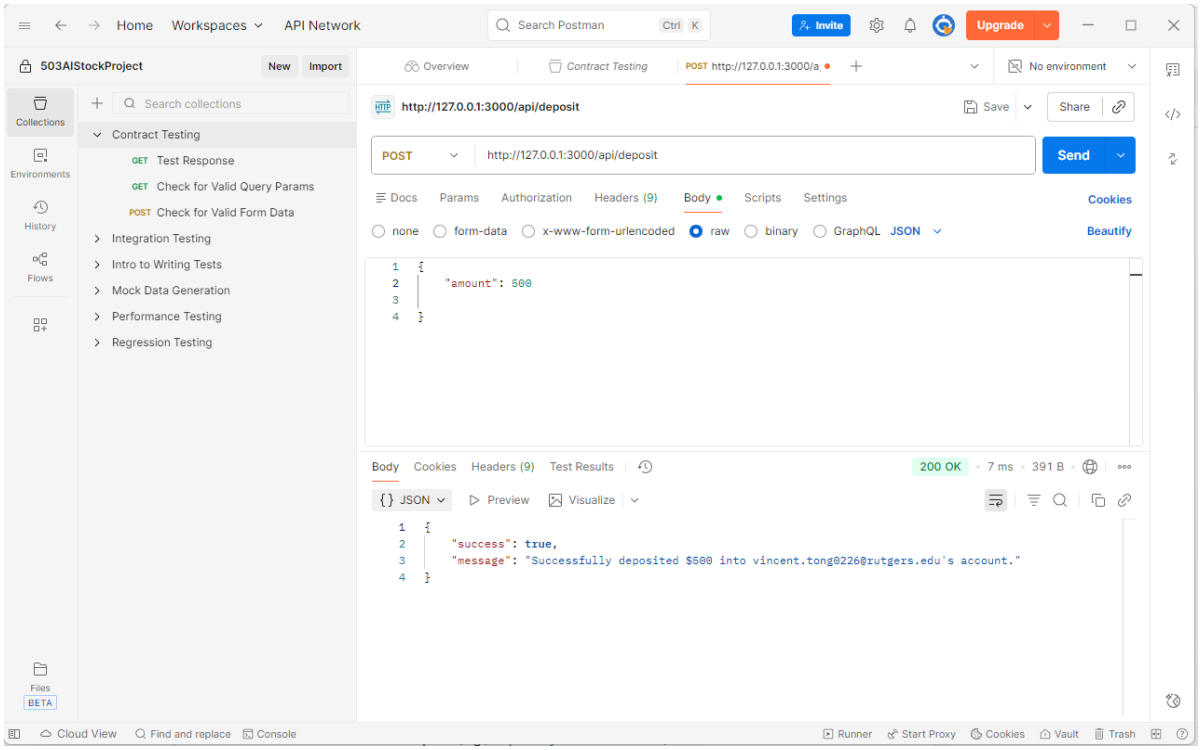
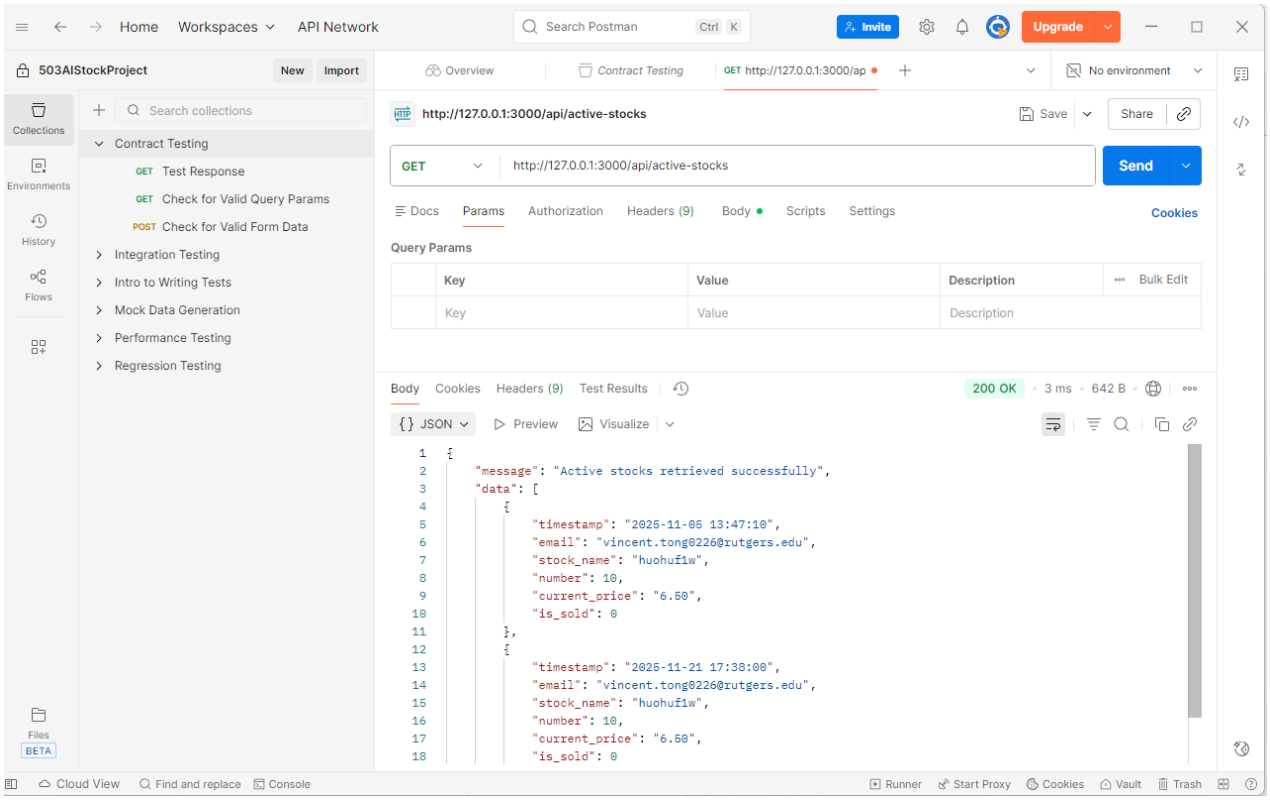


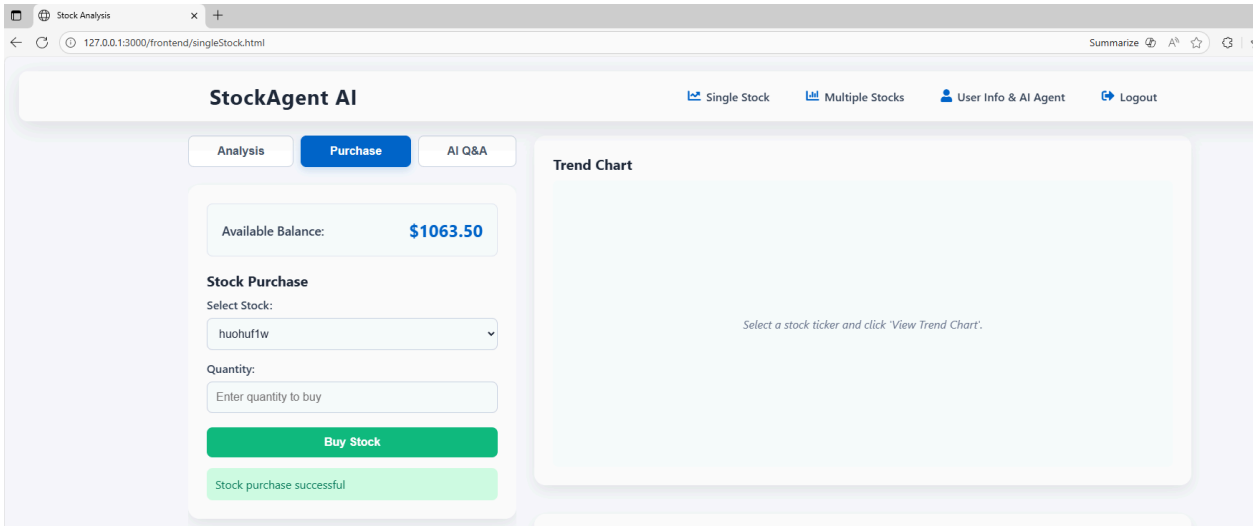
Figure 5.1.2 Deposit Money Tests using Postman

- **Stock Transaction History Retrieval Test**: This validates the functionality for a user to retrieve their personal trading history.
  - **Procedure:** Execute a **GET** request to the user transaction history endpoint (e.g., `/api/buyAndViewStocks`).
  - **Validation:** Verify that the response returns a list of transactions, and each entry correctly contains the following fields from the `stock_transactions` table: `timestamp`, user's `email(Username)`, `stock_name`, `number` of shares, `current_price`, and the `is_sold` status (either 'sold' or 'not sold').

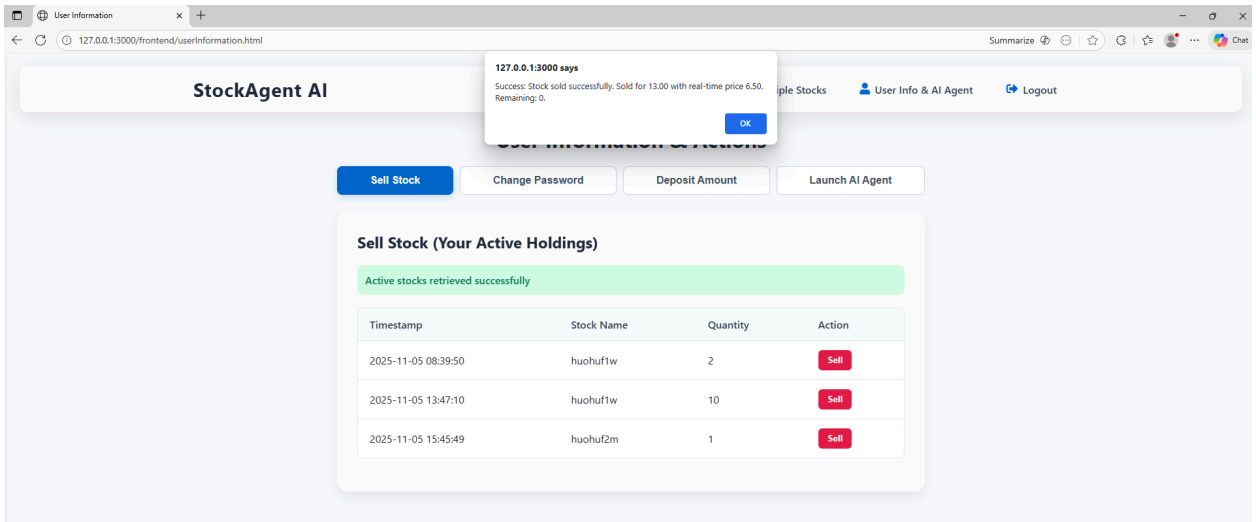


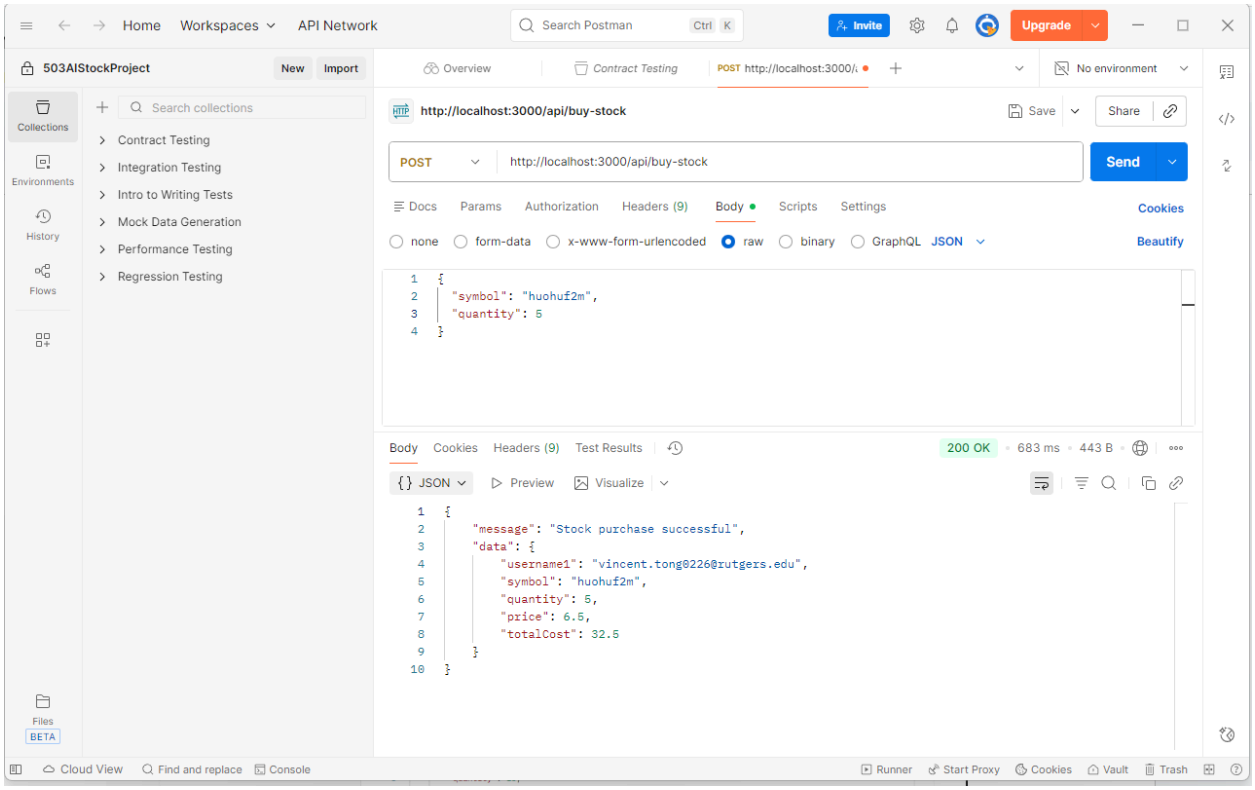
5.1.3 Stock History Transaction Test using Postman

- **Buy Stock Transaction Test:** This test is updated to include the specific success message and required data fields.
    - **Procedure:** Execute a POST request to the buying endpoint (e.g., `/api/buyAndViewStocks`) with the stock symbol and quantity.
    - **Validation 1 (API Response):** Verify the response is `200 OK`, includes the message "Stock purchase successful", and returns the following transactional data fields: username (email), stock's name (symbol), quantity, price, and total cost. Users should also see the message “Stock purchase successful” on their end from the program’s user interface.
    - **Validation 2 (Database Check):** Verify the user's `balance` in the `Users` table has decreased, and a new record is created in the `stock_transactions` table with transaction details and `is_sold` set to 0 (False).



- **Sell Stock Transaction Test:** A **POST** request to the selling endpoint (e.g., `/api/sellStock`) must also perform two actions:
    - **Procedure:** Execute a **POST** request to the dedicated sell endpoint (e.g., /sell-stock) with the stock symbol and quantity.
    - **Validation 1 (API Response):** Verify that the response is 200 OK and that the prompt message includes the exact structure: "Stock sold successfully. Sold for $\$ xxx$ with real-time price xxx. Remaining: xx." The raw API response in Postman will return the specific fields: **"**Stock sold successfully**"**, the amount sold, realTimePrice, remaining quantity, and the boolean status isSold: true.
    - **Validation 2 (Database Check):** Verify the user's `balance` has increased by the sale proceeds. The corresponding record in the `stock_transactions` table must be updated by setting the `is_sold` flag to **1** (True) to reflect the closure of the position.





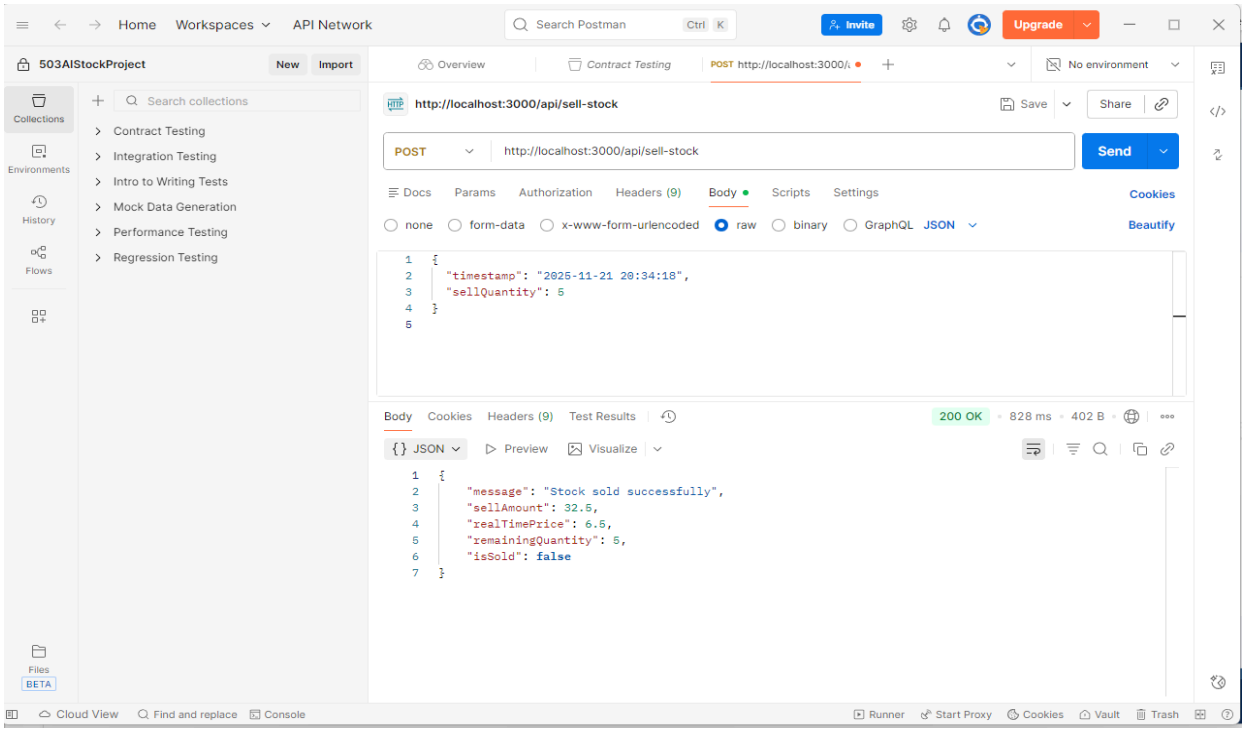


Figure 5.1.4 Buy Stock Test (above) and Sell Stock Test (below) using Postman

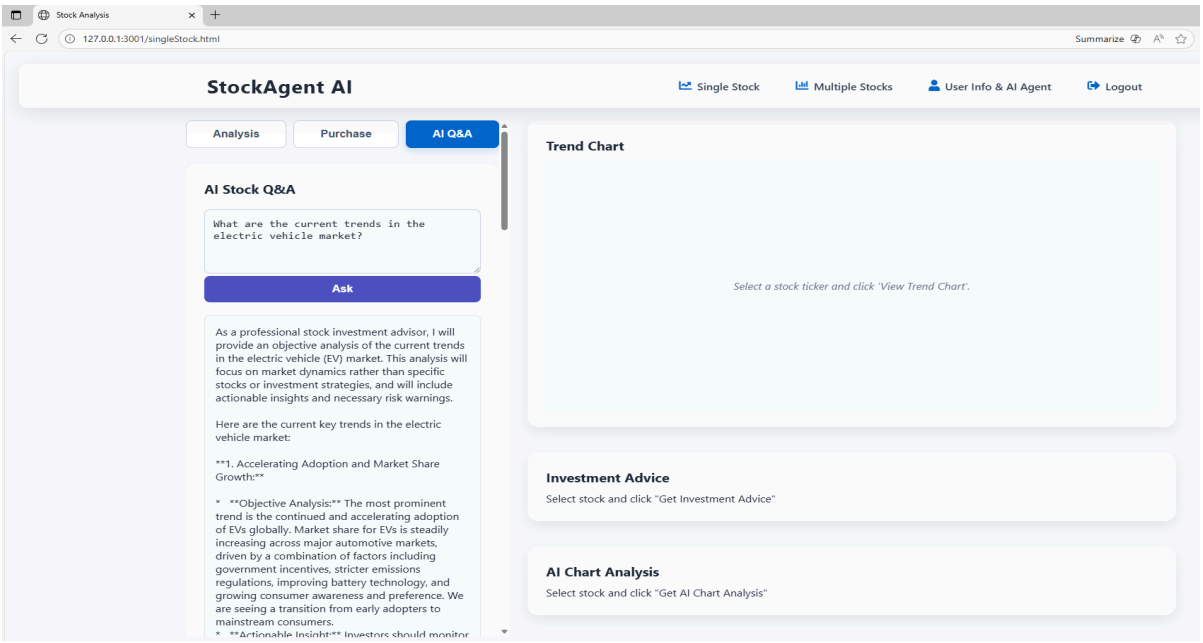
5.2 AI Q&A Performance and Investment Testing

The primary goal of this evaluation is to verify the successful operation of the **AI Agent** within the application. Specifically, we aim to confirm that the agent correctly processes natural language queries and delivers personalized, coherent, and contextually relevant stock investment advice. This verification will focus on the dedicated **AI Q&A and Investment API endpoints** (`/api/stockQA` and the personalized advice `/api/aiPersonalAdvice`) that achieve this functionality.

- **stockQA (Intelligent Stock Question Answering)**

The `stockQA` route is designed for **general, natural language queries** about the market and financial data.

  - **Function:** This route takes complex, unconstrained natural language questions from the user (e.g., "What are the current trends in the electric vehicle market?") and passes them to the AI Agent for processing.
  - **Purpose:** It allows the system to answer a broad range of factual, data retrieval, and analytical questions about individual stocks or market segments. This fulfills the goal of developing an **Intelligent Q&A** feature.
  - **Response:** The AI is expected to return a direct, concise, and coherent text response to the user's question.



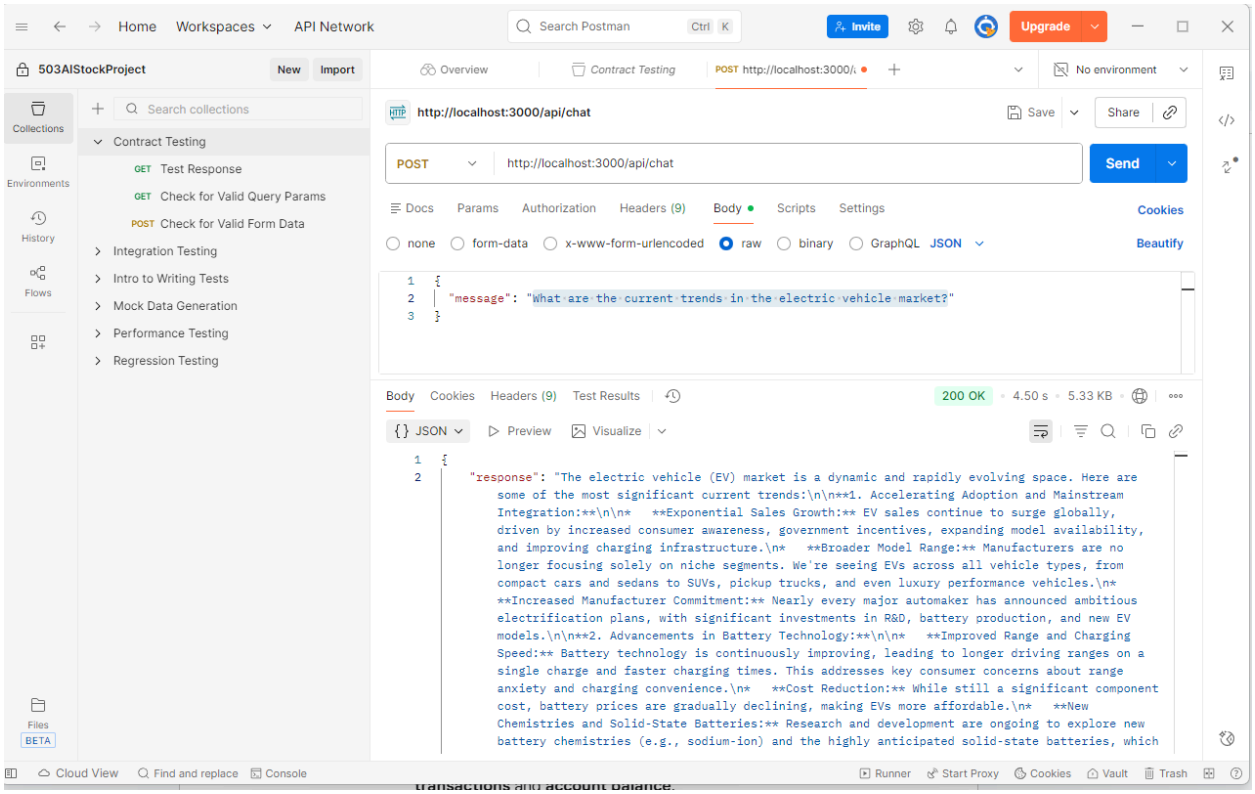


Figure 5.2.1 AI Q&A Test with Postman

- **aiPersonalAdvice (AI-generated Advice Based on User's Holdings)**

The `/api/ai-personal-advice` route is designed to aggregate a user's transaction history and current balance from the database, feeding this contextual data to the AI Agent to generate personalized, tailored investment recommendations.

- **Function**: This confirms the entire system pipeline is robust and delivers the expected result.
  - End-to-End Flow**:** Verify the full pipeline. A client request initiates the process, which triggers a database query for the user's transactions and balance. This data is used for prompt generation, leading to a successful AI API call, which then returns a **Status 200 OK** response with the advice.
  - Input Validation**:** Confirm correct error handling for requests missing the `email` parameter (**Status 400 Bad Request**) and non-existent users (**Status 404 Not Found**).
  - API Resilience**:** Test by simulating a Gemini API failure. The function must gracefully log the error internally while preventing sensitive details from reaching the client, returning a generic

Status 500 Internal Server Error.

- **Purpose Validation:** This stage ensures the function delivers on its promise of providing personalized, actionable investment advice.
    - Contextual Relevance: Assess if the generated advice directly references and interprets the user's data (e.g., matching advice to volatile trading history vs. long-term holdings).
    - Persona Adherence: Verify that the output is professional, non-promotional, and strictly adheres to the requested English advisory tone.
- **Response:** This stage confirms the final output is safe for the end-user and technically correct, protecting both the user's data and the system's infrastructure.
    - Security (**Data Leakage**)**:** Ensure that all error responses are generic (e.g., "Failed to generate advice") and **do not expose internal details**, such as file paths or SQL error codes.
    - Prompt Fidelity**:** Verify (via logs) that the full, accurate dataset (including transactions and balances) is successfully loaded into the AI prompt.
    - Format Integrity**:** Confirm that successful responses are structured as clear, parsable **valid JSON** with readable advice text.

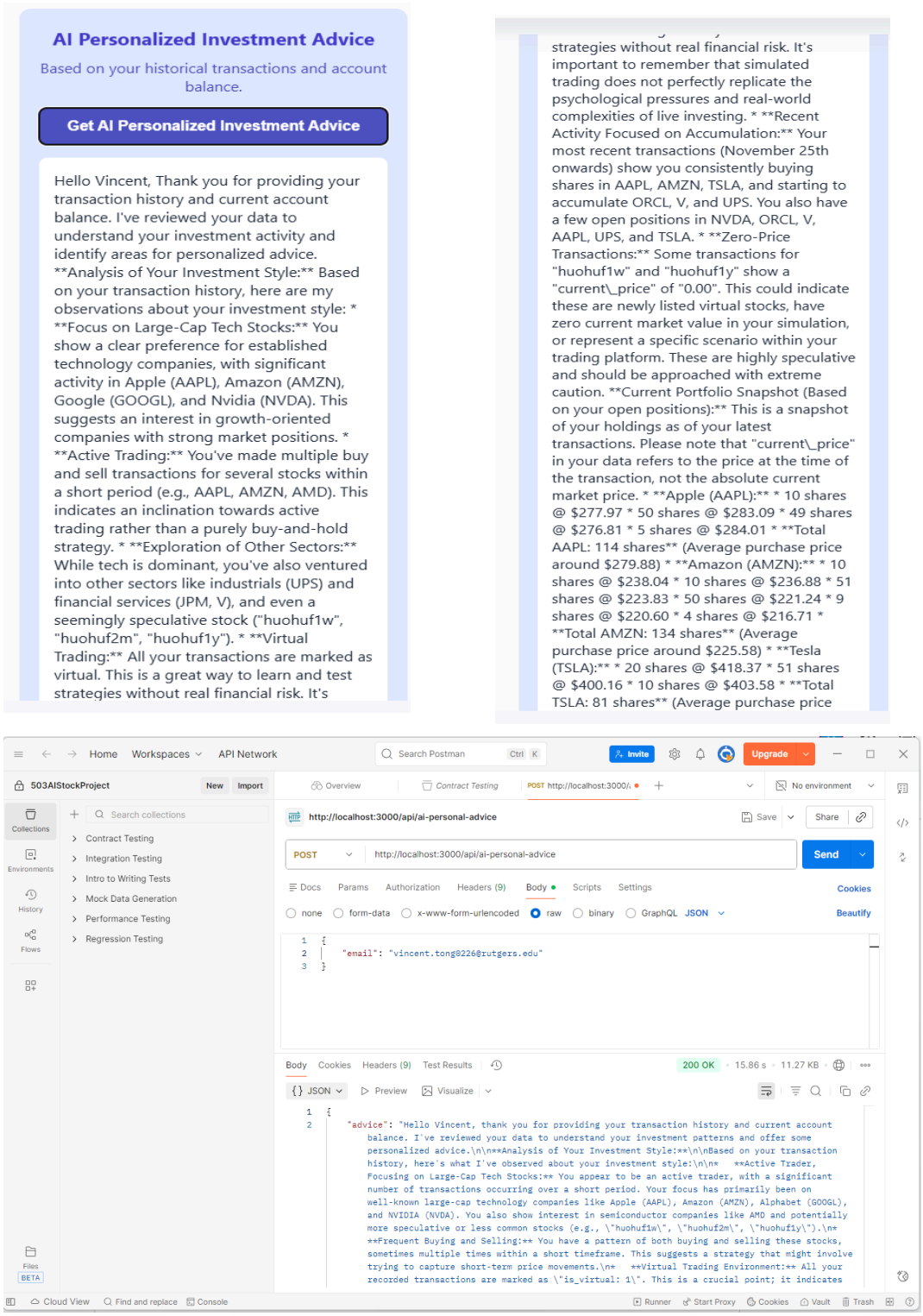
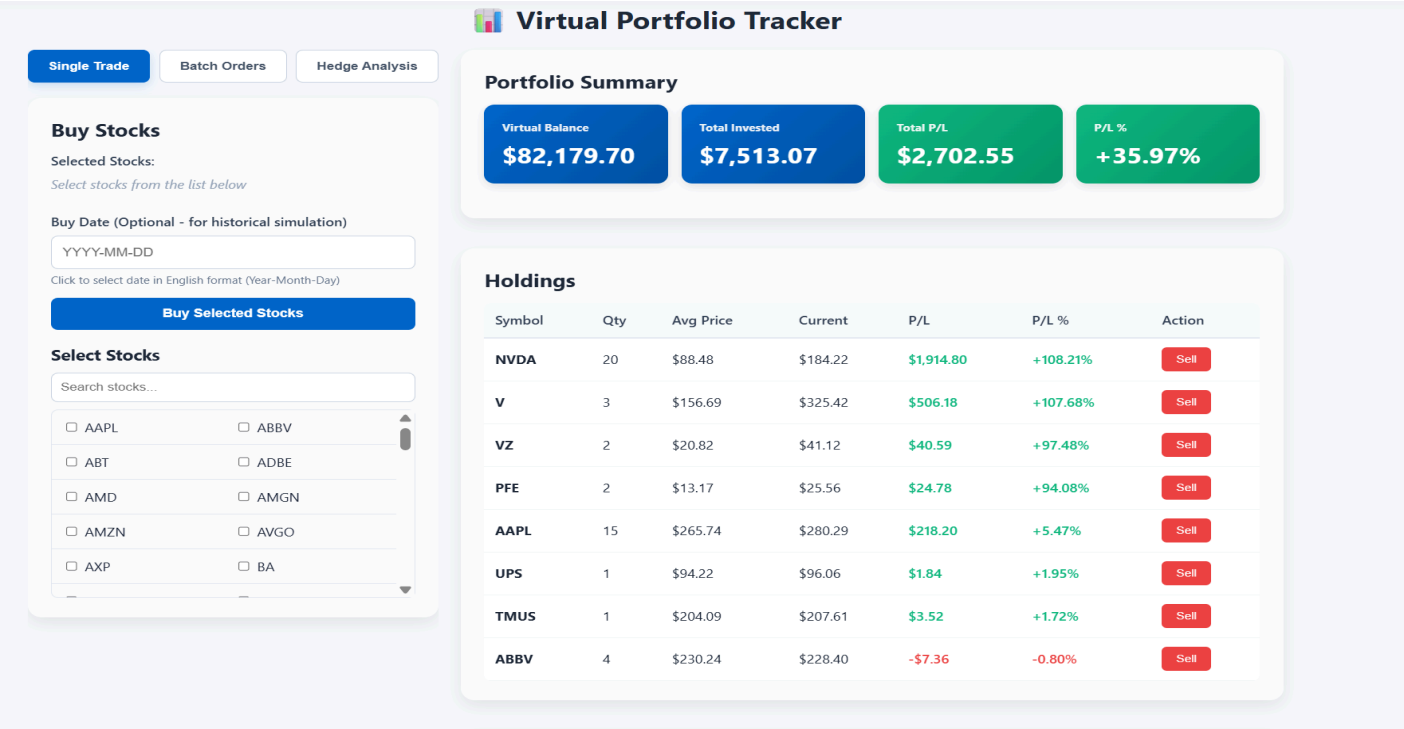


Figure 5.2.2 AI Personalized Investment Advice Test with Postman

5.3 Virtual Portfolio Transaction Tests

- **Virtual Buy Stock Transaction Test**
  - A POST request to the virtual buying endpoint (/api/virtual-buy) must perform two actions:
  - **Procedure**: Execute a POST request to the dedicated virtual buy endpoint (/api/virtual-buy) with the stock symbol, quantity, and optional buyDate for historical simulation
  - **Validation 1 (API Response)**: Verify that the response is 200 OK and that the prompt message includes the exact structure: "Virtual stock purchase successful. Bought xx shares of SYMBOL at $xxx. Total cost: $xxx. Remaining virtual balance: $xxx." The raw API response in Postman will return the specific fields: "success": true, "message": "Virtual stock purchase successful", the symbol, quantity purchased, current price, totalCost, and remainingVirtualBalance.
  - **Validation 2 (Database Check)**: Verify the user's **virtual_balance** in the users table has decreased by the purchase amount. The corresponding record in the stock_transactions table must be inserted with **is_virtual** flag set to 1 (True) and **is_sold** flag set to 0 (False) to indicate an active virtual position.



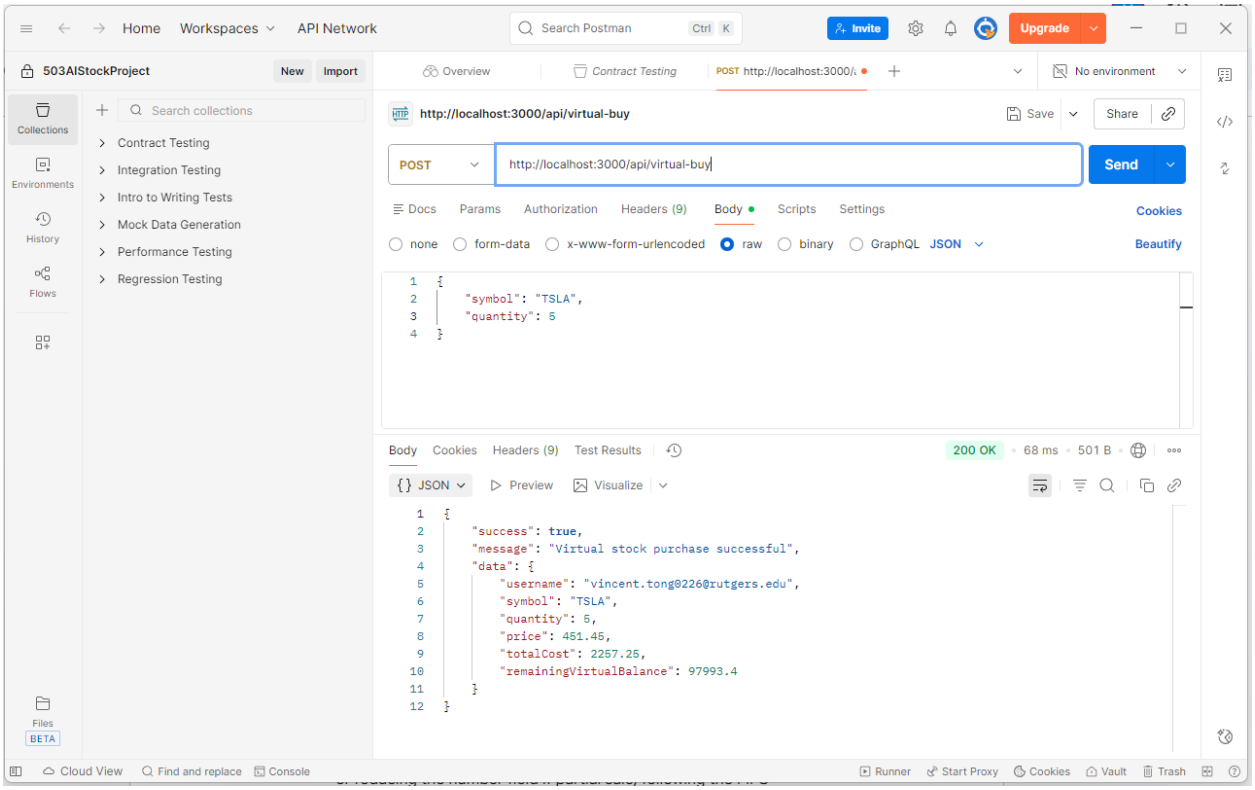


Figure 5.3.1 Virtual Buy Stock Transaction Test with Postman

- **Virtual Sell Stock Transaction** Test
  - A POST request to the virtual selling endpoint (/api/virtual-sell) must perform two actions:
  - **Procedure**: Execute a POST request to the dedicated virtual sell endpoint (/api/virtual-sell) with the stock symbol, quantity, and optional sellDate for historical simulation.
  - **Validation 1 (API Response)**: Verify that the response is 200 OK and that the prompt message includes the exact structure: "Virtual stock sold successfully. Sold xx shares at $xxx. Sale amount: $xxx. Profit: $xxx." The raw API response in Postman will return the specific fields: "success": true, "message": "Virtual stock sold successfully", the symbol, quantity sold, current price, sellAmount, buyCost, and calculated profit.
  - **Validation 2 (Database Check)**: Verify the user's virtual_balance in the users table has increased by the sale proceeds. The corresponding record in the stock_transactions table must be updated by setting the is_sold flag to 1 (True) or reducing the number field if partial sale, following the FIFO (First-In-First-Out) principle to reflect the closure or reduction of the position.

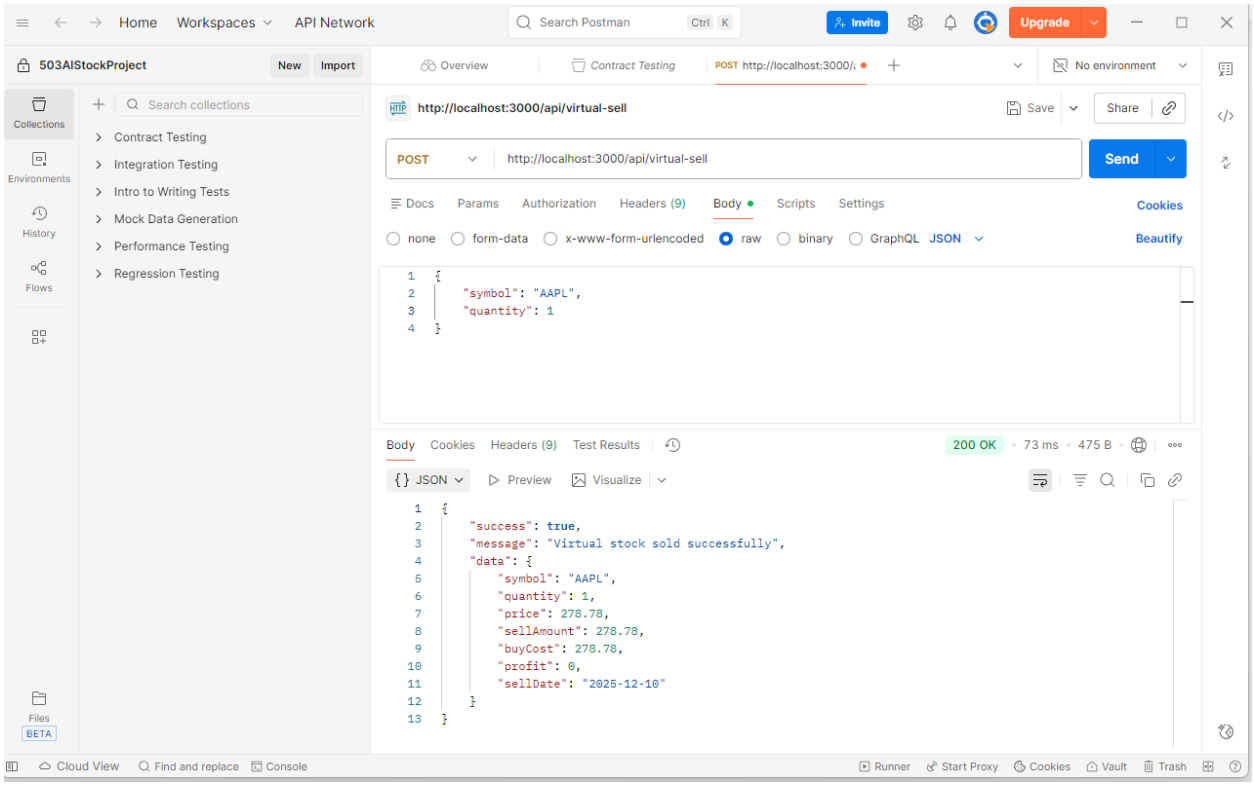
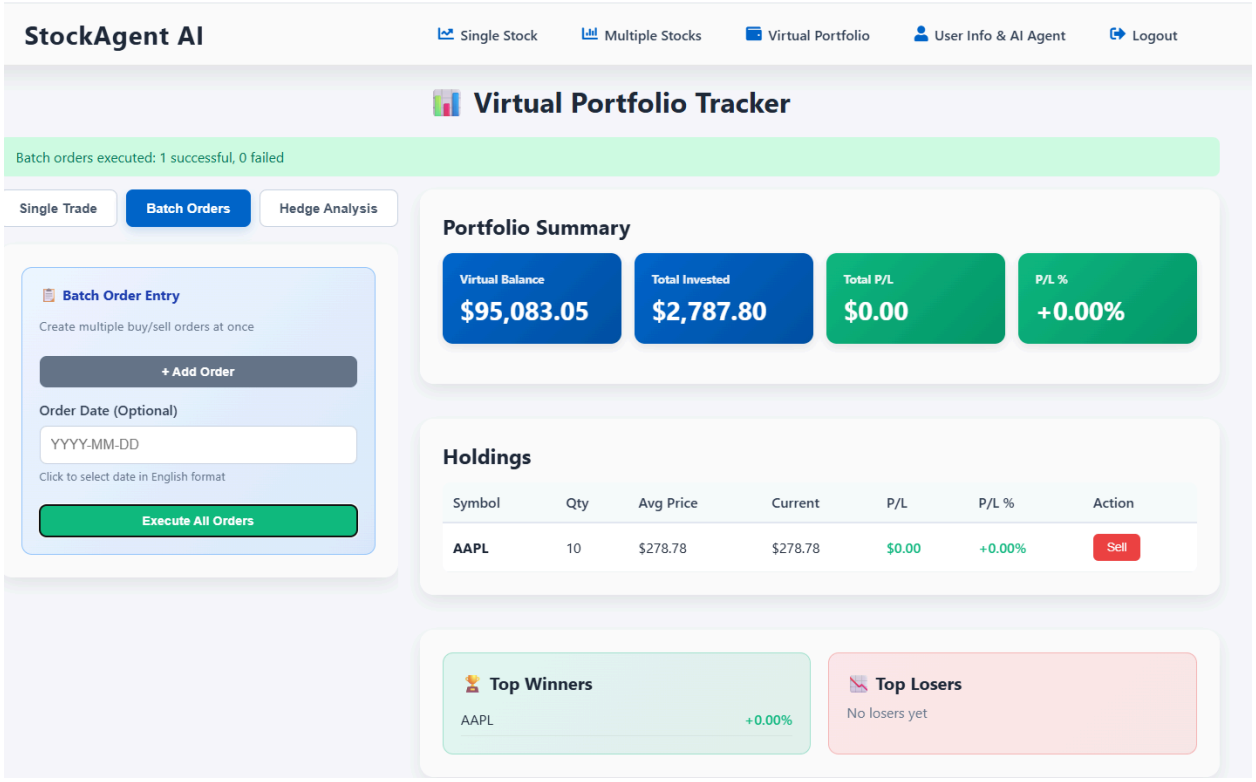


Figure 5.3.2 Virtual Sell Stock Transaction Test with Postman

- **Batch Buy Transaction** Test
  - A POST request to the batch buying endpoint (/api/batch-buy) must process multiple purchases:
  - **Procedure**: Execute a POST request to the batch buy endpoint (/api/batch-buy) with an orders array containing multiple stock symbols and quantities, plus an optional buyDate.
  - **Validation 1 (API Response)**: Verify that the response is 200 OK and includes a results array with individual outcomes. Each result includes: symbol, success status (boolean), and either transaction details (quantity, price, totalCost) for successful orders or error messages for failed ones. The summary field shows: total orders, successful count, and failed count.
  - **Validation 2 (Database Check)**: Verify that for each successful order in the batch, the user's virtual_balance has decreased by the respective totalCost, and new records with is_virtual=1 and is_sold=0 are inserted into the stock_transactions table. Failed orders must not affect the database state.



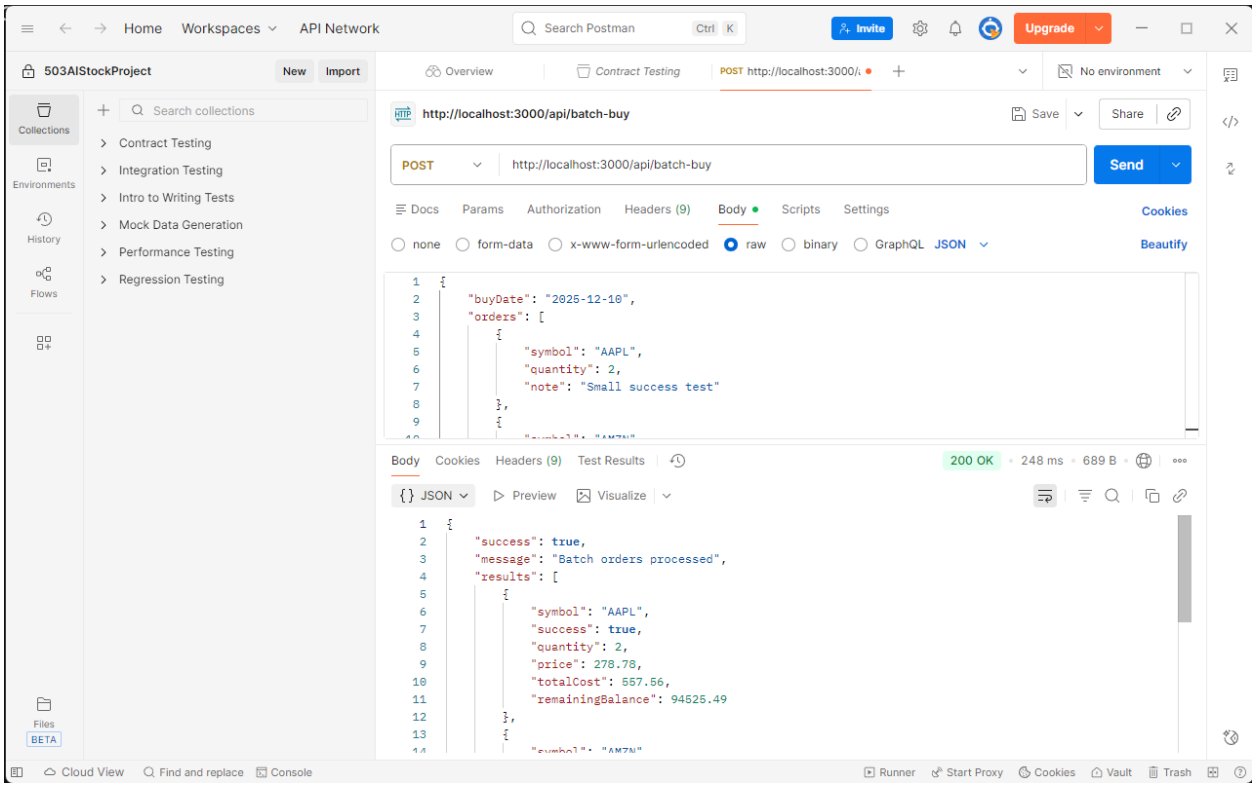
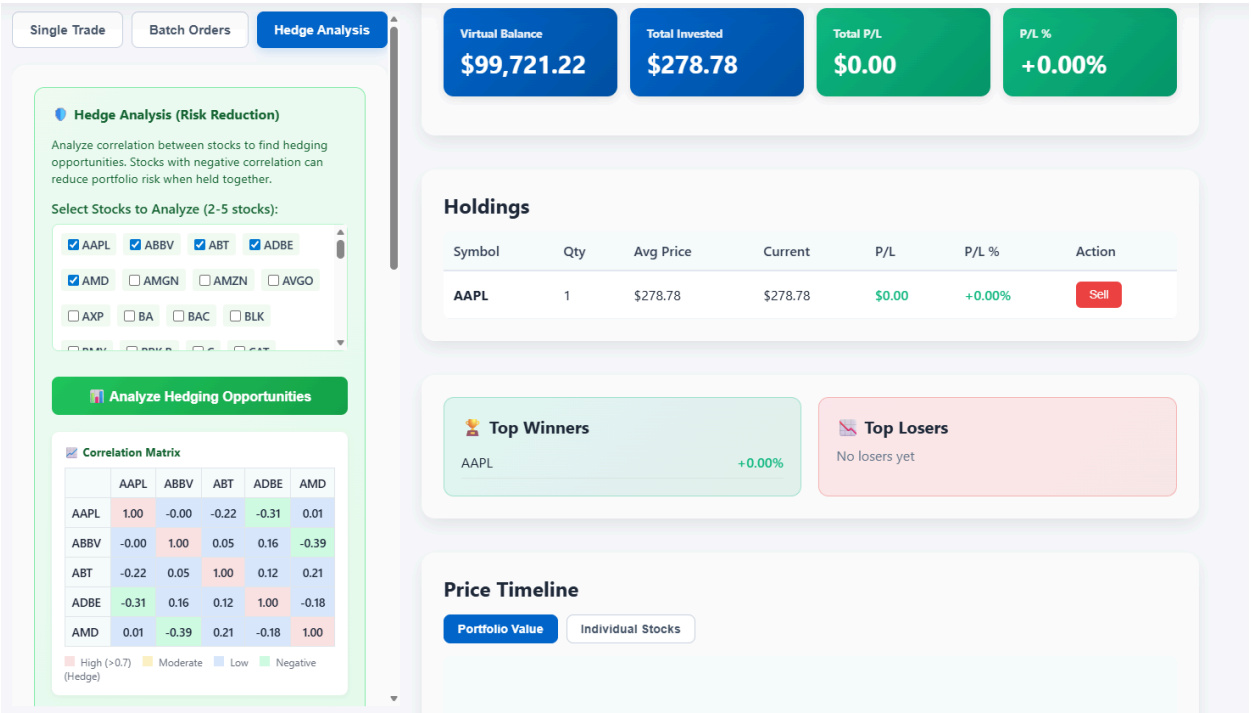
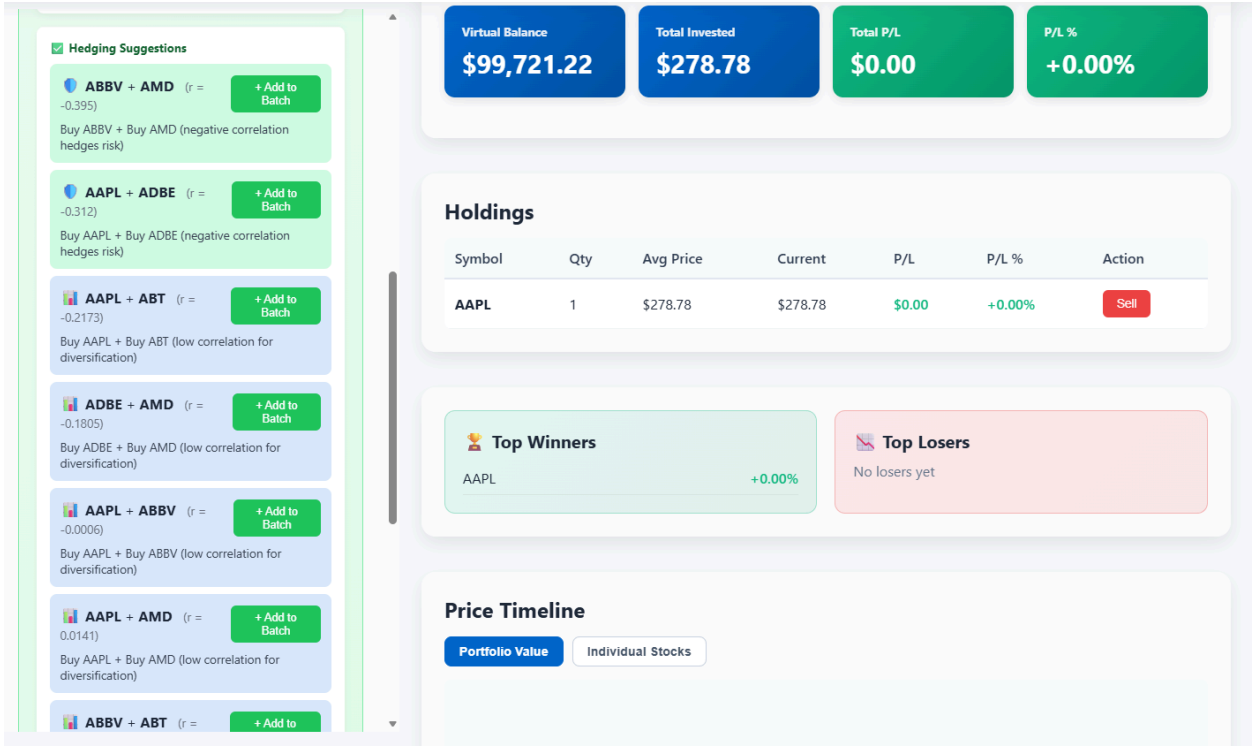
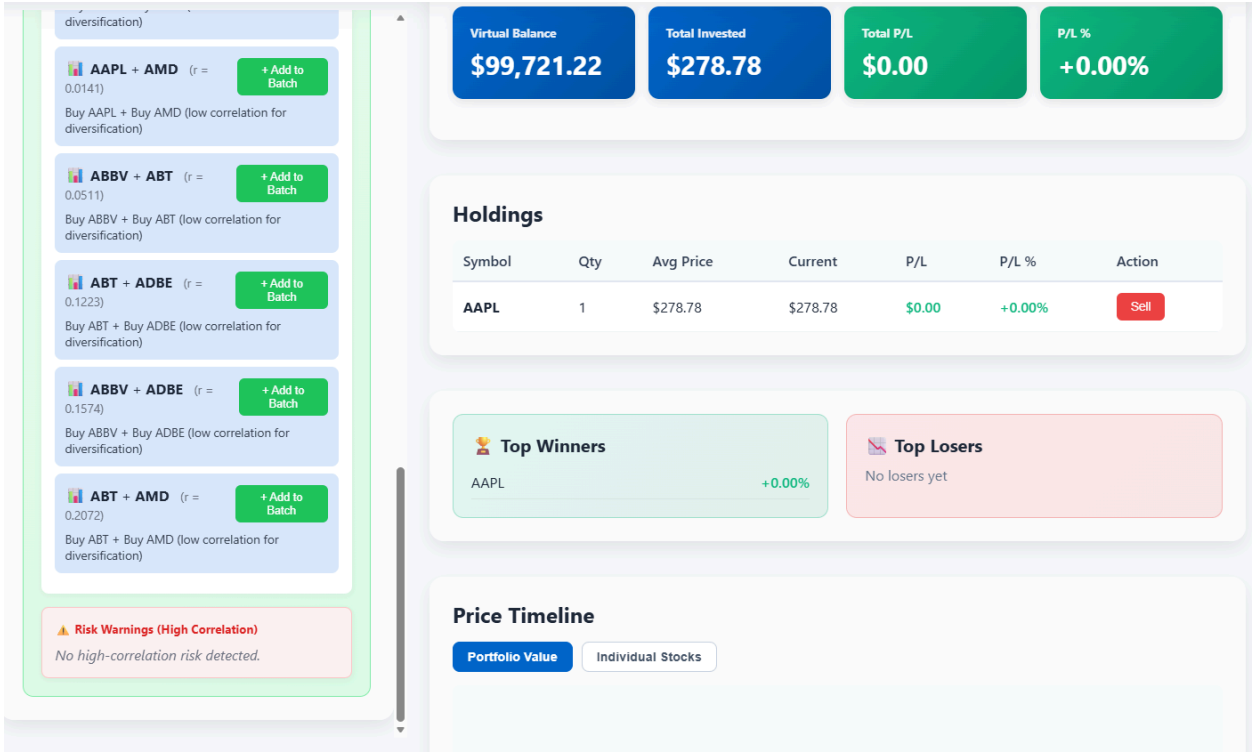


Figure 5.3.3 Batch Buy Stock Transaction Test with Postman

- Hedge Analysis Test
    - A POST request to the hedge analysis endpoint (/api/hedge-analysis) must calculate correlations.
    - **Procedure**: Execute a POST request to the hedge analysis endpoint (/api/hedge-analysis) with a symbols array containing 2-5 stock ticker symbols.
    - **Validation 1 (API Response)**: Verify that the response is 200 OK and includes the analysis object with exact structure: correlationMatrix (symmetric NxN matrix with values between -1.0 and 1.0), hedgeSuggestions array (each containing stockA, stockB, correlation, type, and suggestion text), riskWarnings array (for high-correlation pairs with warning messages), and summary statistics (totalPairs, negativePairs, lowPairs, highPairs counts)
    - **Validation 2 (Data Accuracy)**: Verify correlation calculation accuracy: diagonal elements equal 1.0, matrix symmetry (correlationMatrix[A][B] = correlationMatrix[B][A]), proper classification of pairs (negative: r < -0.3, low: -0.3 ≤ r < 0.3, moderate: 0.3 ≤ r < 0.7, high: r ≥ 0.7), and hedgeSuggestions are sorted with most negative correlations first.




Hedging Suggestions
ABBV + AMD (r = -0.395)
+ Add to Batch
Buy ABBV + Buy AMD (negative correlation hedges risk)
AAPL + ADBE (r = -0.312)
+ Add to Batch
Buy AAPL + Buy ADBE (negative correlation hedges risk)
AAPL + ABT (r = -0.2173)
+ Add to Batch
Buy AAPL + Buy ABT (low correlation for diversification)
ADBE + AMD (r = -0.1805)
+ Add to Batch
Buy ADBE + Buy AMD (low correlation for diversification)
AAPL + ABBV (r = -0.0006)
+ Add to Batch
Buy AAPL + Buy ABBV (low correlation for diversification)
AAPL + AMD (r = 0.0141)
+ Add to Batch
Buy AAPL + Buy AMD (low correlation for diversification)
ABBV + ABT (r =
+ Add to
Virtual Balance
$99,721.22
Total Invested
$278.78
Total P/L
$0.00
P/L %
+0.00%
Holdings
Symbol
Qty
Avg Price
Current
P/L
P/L %
Action
AAPL
1
$278.78
$278.78
$0.00
+0.00%
Sell
Top Winners
AAPL
+0.00%
Top Losers
No losers yet
Price Timeline
Portfolio Value
Individual Stocks


diversification)
AAPL + AMD (r = 0.0141)
+ Add to Batch
Buy AAPL + Buy AMD (low correlation for diversification)
ABBV + ABT (r = 0.0511)
+ Add to Batch
Buy ABBV + Buy ABT (low correlation for diversification)
ABT + ADBE (r = 0.1223)
+ Add to Batch
Buy ABT + Buy ADBE (low correlation for diversification)
ABBV + ADBE (r = 0.1574)
+ Add to Batch
Buy ABBV + Buy ADBE (low correlation for diversification)
ABT + AMD (r = 0.2072)
+ Add to Batch
Buy ABT + Buy AMD (low correlation for diversification)
Risk Warnings (High Correlation)
No high-correlation risk detected.
Virtual Balance
$99,721.22
Total Invested
$278.78
Total P/L
$0.00
P/L %
+0.00%
Holdings
Symbol
Qty
Avg Price
Current
P/L
P/L %
Action
AAPL
1
$278.78
$278.78
$0.00
+0.00%
Sell
Top Winners
AAPL
+0.00%
Top Losers
No losers yet
Price Timeline
Portfolio Value
Individual Stocks

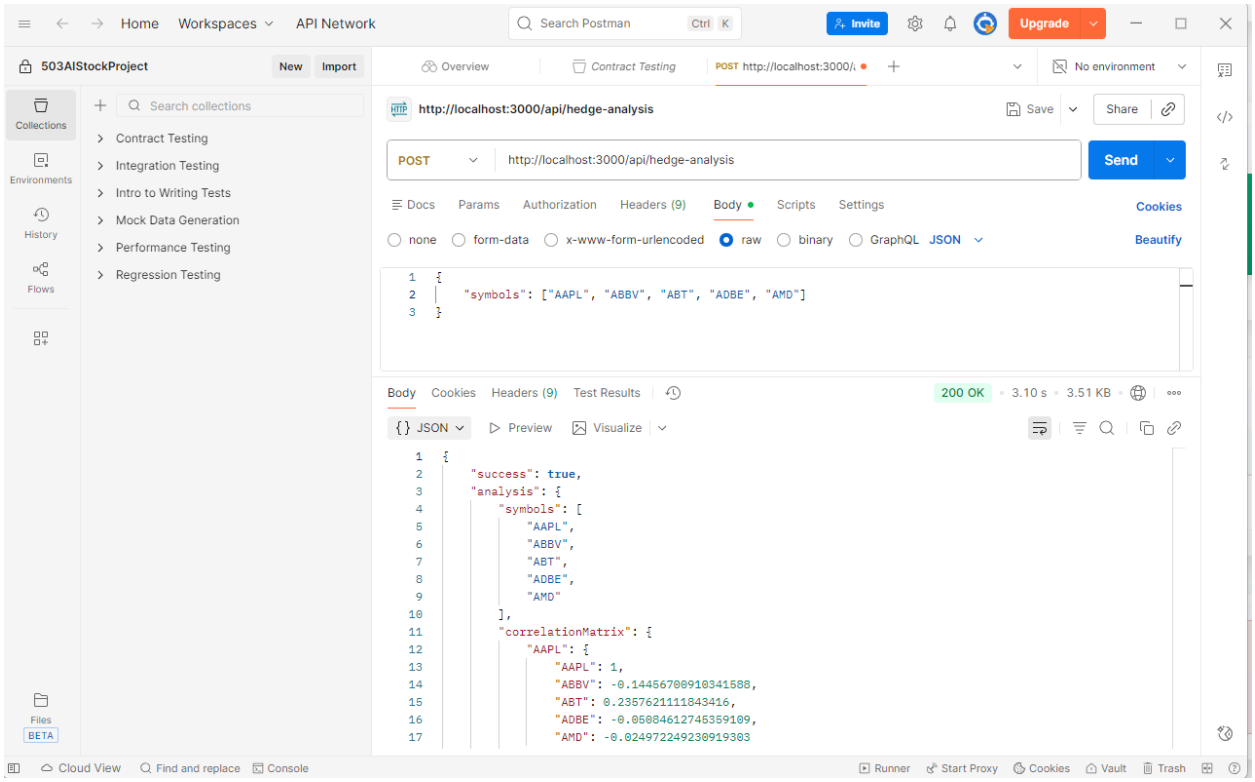


Figure 5.3.4 Hedge Analysis Test with Postman

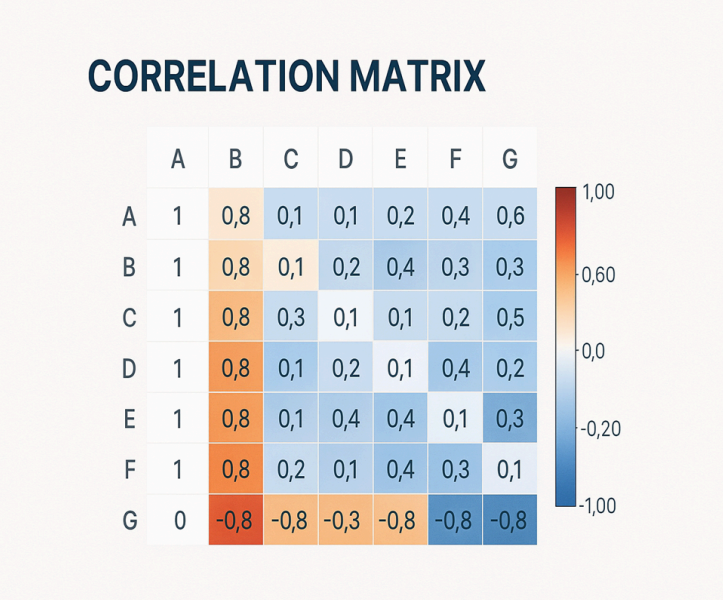
5.4 System Stability and Mathematical Logic Testing

This phase confirms the accuracy of the financial models central to the Multiple Stock Module.

- **Correlation Calculation Accuracy**
  - **Procedure:** Run the Multiple Stock Module with historical data for multiple stocks.
  - **Validation:** Manually calculate the correlation coefficient between any two stocks using the formula:

$$\text{Correlation}(A, B) = \frac{\text{Cov}(A, B)}{\sqrt{\text{Var}(A) \times \text{Var}(B)}}$$

Compare this result with the value displayed in the **Correlation Matrix** to ensure the system's logic for calculating covariance and variance is sound.



- If two stocks are highly correlated, their price changes may be similarly affected by the same market event, and risk cannot be spread.
- If two stocks are negatively correlated, price movements can be partially hedged, reducing the volatility of the portfolio.

- **Portfolio Weight Calculation, Normalization, and Monitoring**

The overall goal of this process is to determine the fraction of the total portfolio value that should be allocated to each stock, based on its calculated risk and potential return (as measured by the Sharpe ratio) and its relationship with other stocks (as indicated by correlation).

- **Initial Weight Calculation and Adjustments**

The system first calculates an **Initial Weight** for each stock by adjusting it based on its individual risk and risk-adjusted return (Sharpe ratio).

- Risk-based Adjustment: Risk Adjustment = risk / total risk
- Sharpe Ratio Adjustment: SharpeAdjustment = 1 + (sharpeRatio / 2)
- Combined Initial Weight: InitialWeight = RiskAdjustment * SharpeAdjustment

- **Correlation Adjustment**

Next, the initial weights are adjusted based on the Correlation Matrix.

- **Logic:** Stocks that are **highly correlated** will have their weights **reduced** because they do not effectively spread risk. Stocks that are **negatively correlated** offer a hedging benefit, reducing the portfolio's volatility.

- **Final Normalization**

After the initial and correlation adjustments, the sum of all calculated weights will generally not equal 1. The Normalization step ensures the total allocation equals 100%.

- **Procedure:** The system takes the final adjusted weight for each stock and divides it by the sum of all adjusted weights.
- **Validation**: The system must ensure the sum of all weights equals **1**.
- **Final Normalized Weight of Stock i** = Adjusted Weight of Stock i / Sum of All Stocks' Adjusted Weight
- This ensures that **100%** of the designated investment capital is allocated across the portfolio.

- **Integration with Portfolio Monitoring**

The **Portfolio Monitoring** system takes these final, fixed weights (and the resulting purchased shares) as the investment baseline and runs continuously to track changes.

- **Monitoring Mechanism:** The app polls for the **Current Real-Time Price** for every stock symbol.
- **Performance Calculation:** The system calculates the individual stock performance as a percentage:

  Performance (%) = (Current Price - Initial Purchase Price) / Initial Purchase Price * 100%

- **Notification and Highlighting:** The system uses the calculated Performance (%) to generate a brief summary, showing the **Overall Net Gain/Loss** and clearly highlighting the **Winners** (those with the highest positive Performance (%)) and Losers (those with the lowest negative Performance (%)).

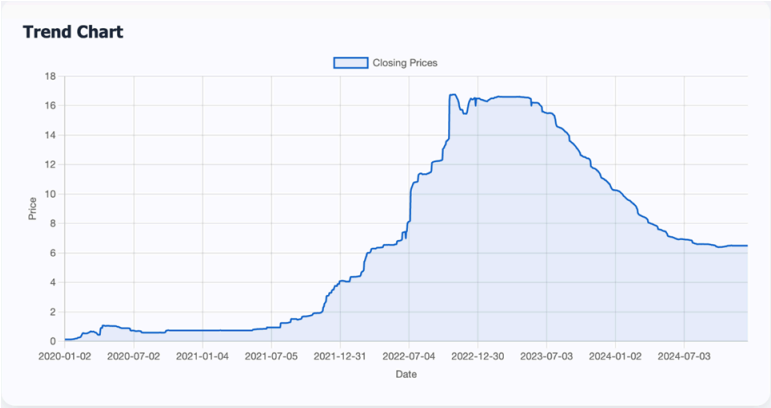
- **Trend Chart Accuracy**

This test confirms that the application's visual representation of historical price data accurately and functionally mirrors the underlying financial data used for the Single Stock Analysis. The chart is a critical component, as it provides the visual context that leads to the

derived **Risk Analysis** metrics, such as maximum drawdown and Volatility.

The evaluation process involves two key steps:

- **Data Integrity Check:**
  - **Procedure:** Obtain the raw historical price data used by the system (e.g., the data points feeding the chart)
  - **Validation:** Verify that the **Closing Prices** plotted on the chart accurately match the raw data points used for the corresponding dates.
- **Functionality Check (User Interaction):**
  - **Procedure:** Manually interact with the chart interface itself.
  - **Validation:** Confirm that the user can successfully **click on a specific date** on the visual trend line to correctly retrieve and display the **Closing Price** for that exact date, validating the chart's required interactive functionality.



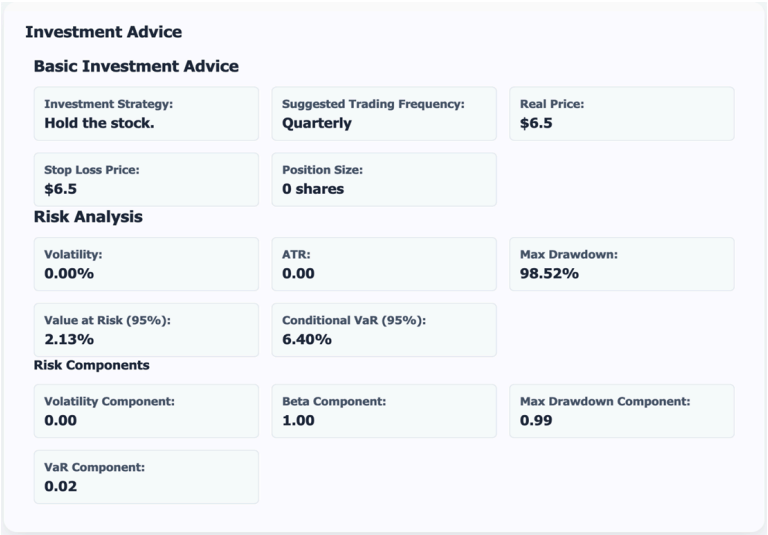


Figure 5.4 Stock Market Trends and Risk Analysis shown on the singleStock.html

5.5 AI Agent Simulation and Behavioral Analysis Testing

This testing phase is designed to evaluate the AI Agent's behavioral logic and decision interpretability in a simulated trading environment using mock data analysis. Figure 5-2 illustrates an AI-driven simulated trading system that tracks transactional actions and agent-generated commentary to verify the transparency of its decision-making mechanism.

- **Test Findings:** The AI Agents demonstrated distinct trading personas based on their market interpretation and risk preferences:
    - **Agent 1 (Strategic & Fundamentalist):**
        - **Strategy:** Demonstrated a strategy of accumulation followed by strategic rebalancing based on fundamental news.
        - **Behavior:** Aggressively accumulated **Stock B** during Day 1 and early Day 2 (executing buy orders between 205.00 and 207.00). However, the agent's "Live Event Feed" commentary revealed a shift in focus toward **Stock A**, citing "new leadership" and "strategic shifts" as a "turnaround story."
        - **Conclusion:** Successfully exhibited a complex persona that separates high-frequency trading actions (buying B) from long-term value assessment (analyzing Stock A's recovery potential).
    - **Agent 0 (Risk-Averse / Bearish):**
        - **Strategy:** Showcased a decisive, risk-off approach to volatility or stagnating prices.
        - **Behavior:** Executed significant sell orders for **Stock B** on Day 2, liquidating 9,000 shares at the 200.00 price point. This suggests the agent identified a resistance level or a negative trend in Stock B and prioritized liquidity over holding.
        - **Conclusion:** The agent successfully demonstrated a protective exit strategy, rapidly reducing exposure to a specific asset (Stock B) to preserve capital.

This test confirms that the AI Agents not only execute trades but can also generate human-readable analysis, integrating technical execution with fundamental reasoning (as seen in the forum post), thereby meeting the system's goal of developing "human-like trading personas under AI governance."

- **System Integration**

The AI agent is designed to operate seamlessly within a larger ecosystem through RESTful APIs. These interfaces allow external modules—such as user dashboards, transaction processors, or notification systems—to interact with the agent by querying its state, triggering trades, or receiving updates. The system also implements centralized logging and real-time monitoring for every transaction and AI decision. This enables administrators to identify

anomalies, track performance metrics, and debug errors through structured logs. Error-handling and rollback mechanisms further ensure transactional integrity and provide graceful degradation in the event of system failure or API failure.

- **Security Features**

Security in the AI agent is multi-layered:

- **Trade Validation:** The system validates every trade by checking account balance, current holdings, and risk limits (e.g., maximum allocation per stock). These checks ensure users cannot over-leverage or take unbounded risk.
- **Exception Handling:** Exception-handling wrappers are placed around key operations to prevent failures—whether from external APIs, model errors, or database faults—from crashing the system. Fallback logic or alert mechanisms are activated instead.
- **Backup Systems:** Periodic storage of transaction and portfolio snapshots prevents data loss and enables recovery.

- **Suggested Improvements**

Several enhancements are proposed to boost the system's performance and intelligence:

- **Macro Indicator Integration:** Incorporate market indicators such as GDP, inflation, and sector rotation to enhance macro-level decision support.
- **AI Refinement:** Enhance the AI engine using techniques such as ensemble learning, reinforcement learning, or temporal attention models to boost adaptability and accuracy.
- **Risk Modeling:** Introduce advanced simulations (Monte Carlo or Bayesian estimations) to strengthen risk modeling.
- **Scalability:** For large-scale deployment, performance monitoring dashboards and batch processing pipelines are recommended.

Figure 5.5.1 AI agent configuration parameters page

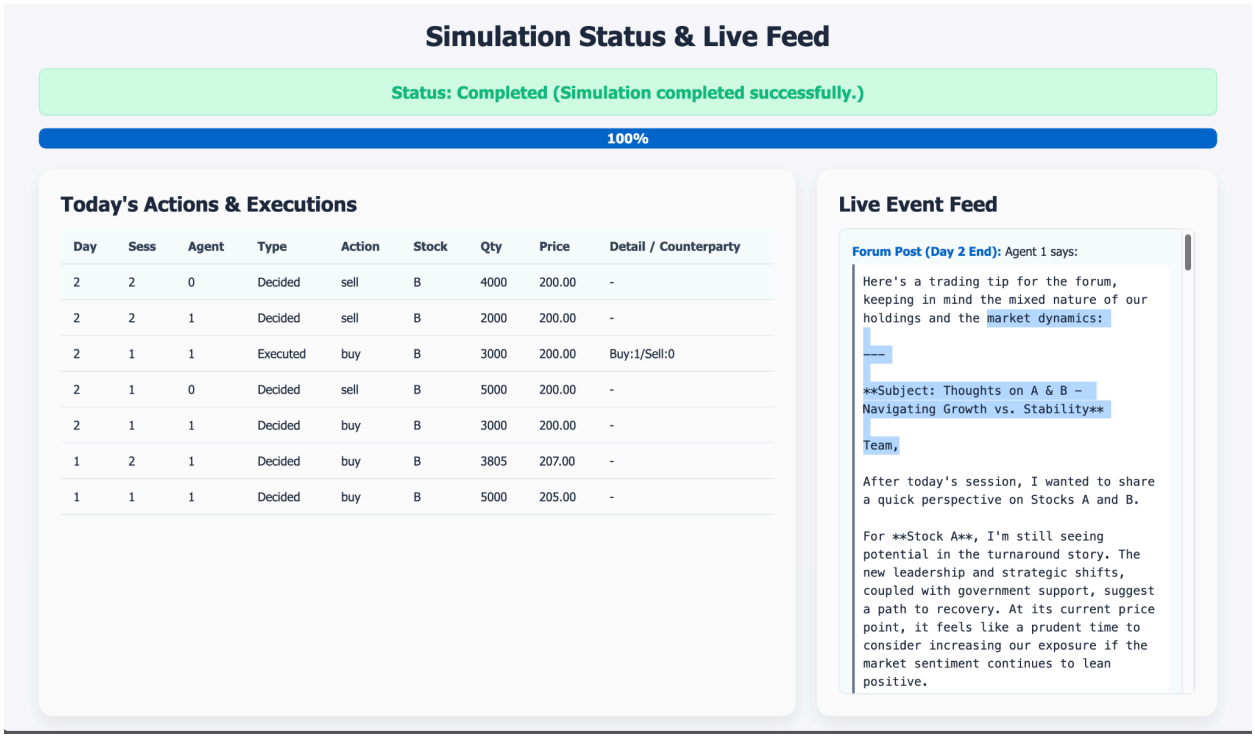


Figure 5.5.2 AI agent analysis page

6. Comparative Analysis: Human vs. AI Implementation

6.1 Evaluation Methodology

To rigorously evaluate the efficacy of AI-driven development in complex financial software, we adhered to the strict constraints defined in the course requirements. The AI Agent (Cursor) was tasked with generating core system modules using **single, non-technical functional prompts** (e.g., "Make a login system"). This output was then rigorously audited and compared against our team's manual implementation, which was engineered with production-grade reliability, security, and scalability in mind.

6.2 Implementation Status Summary

Module Name	Functional Requirement (Prompt Used)	Result	Key Difference
1. User Authentication	"Create a login system where users can register. It must secure passwords and keep users logged in."	Pass	AI lacked session timeouts and used weaker hashing (SHA-256) vs. Human's bcrypt with salt rounds.
2. Batch Order Processing	"Allow users to buy multiple stocks at once. If one fails, the others should still work."	Pass	AI used sequential loops stopping on error; Human used Promise.allSettled for partial success transactionality.
3. Financial Risk Metrics	"Calculate risk numbers like Volatility and Max Drawdown for a list of stock prices."	Pass	AI missed "Annualization" adjustments and crashed on empty/NaN data arrays.
4. Real-time Stock Data	"Fetch current stock prices. If the main source fails,	Fail	AI failed to implement a complex fallback

	try another way to get data."		chain (API -> API -> CSV) and timeout logic; code hung on API failure.
5. Virtual Portfolio	"Manage a $100k virtual balance for trading. Keep it separate from real money."	Pass	AI used a local JSON file for data storage, causing synchronization issues; Human integrated it into the relational MySQL schema.
6. AI Stock Q&A	"Answer user questions about their stocks using an AI model."	Pass	AI gave generic answers; Human injected "Portfolio Context" (current holdings) into the prompt for personalized answers.
7. Hedge Analysis	"Find stock pairs that move in opposite directions to reduce risk."	Pass	AI successfully used Python libraries for correlation, matching human accuracy, but lacked visualization.

6.3 Deep Dive Comparison: Architecture & Logic

This section analyzes the specific architectural and logical divergences between the Human and AI implementations for **all seven modules** listed above.

1. User Authentication & Security

- **Prompt:** "Create a login system where users can register. It must secure passwords and keep users logged in."
    - **Human Implementation:**
        - **Encryption:** Utilized bcrypt with a configurable salt round of 10. This is computation-intensive by design to resist brute-force and rainbow table

attacks.
        - **Session Management:** Implemented server-side sessions with a 30-minute absolute timeout to prevent session hijacking.
    - **AI Agent Implementation:**
        - **Encryption:** Used the standard crypto library with simple SHA-256 hashing. While "hashed," it is too fast and lacks salting, making it vulnerable.
        - **Session Management:** Implemented a simple "keep me logged in" cookie without an expiration policy, posing a security risk on public computers.

2. Robustness in Batch Processing (Transaction Integrity)

- **Prompt:** "Allow users to buy multiple stocks at once. If one fails, the others should still work."
    - **Human Implementation:**
        - **Resilience:** Implemented Promise.allSettled to execute orders in parallel. If buying Stock A fails (e.g., limit exceeded), but Stock B is valid, Stock B will still be processed.
        - **Feedback:** Returns a structured summary array [{id: 1, status: "fulfilled"}, {id: 2, status: "rejected", reason: "Funds"}] to the frontend.
    - **AI Agent Implementation:**
        - **Logic:** Generated a standard for loop using await.
        - **Failure Mode:** If the third order in a batch of ten failed, the loop threw an unhandled exception, causing the script to crash immediately. The remaining seven valid orders were never executed.

3. Financial Accuracy in Risk Management

- **Prompt:** "Calculate risk numbers like Volatility and Max Drawdown for a list of stock prices."
    - **Human Implementation:**
        - **Standardization:** Applied the Root-252 adjustment to convert daily standard deviation into **Annualized Volatility**, which is the industry standard for comparison.
        - **Edge Case Handling:** Added explicit checks for division by zero and NaN values to prevent server crashes when analyzing stocks with insufficient history (e.g., IPOs).
    - **AI Agent Implementation:**
        - **Logic:** Calculated raw standard deviation of the array. While mathematically correct for the sample, it is financially meaningless without annualization.
        - **Reliability:** Assumed perfect input data. Crashed when passed an array with a single price point (Resulting in NaN for deviation).

4. Real-time Stock Data Reliability (Fallback Strategies)

- **Prompt:** "Fetch current stock prices. If the main source fails, try another way to get data."
    - **Human Implementation:**
        - **Multi-Layer Fallback:** Finnhub API (Primary) -> Alpha Vantage (Secondary) -> Local CSV Parser (Emergency).
        - **Timeout Logic:** Implemented a strict 5-second timeout. If the API hangs, the system automatically aborts and switches to the CSV source to prevent page load delays.
    - **AI Agent Implementation:**
        - **Logic:** Used a simple try-catch block. It attempted to fetch data from the API, and if an error occurred, it simply logged "Error fetching data" and returned null.
        - **Failure Mode:** The system failed to implement the logic to *read* the backup CSV file, resulting in a blank dashboard when the API rate limit was reached.

5. Virtual Portfolio Architecture (Data Integrity)

- **Prompt:** "Manage a $100k virtual balance for trading. Keep it separate from real money."
    - **Human Implementation:**
        - **Schema Integration:** Added is_virtual (boolean) and virtual_balance (decimal) columns directly to the Transactions and Users MySQL tables.
        - **Benefit:** Allows unified reporting. We can use a single SQL query to compare a user's Real vs. Virtual performance.
    - **AI Agent Implementation:**
        - **Architecture:** Created a separate file named virtual_data.json to store virtual trades.
        - **Critique:** The "Flat File" approach causes concurrency issues (two users attempting to update the file simultaneously might overwrite it) and renders it impossible to perform complex queries (such as joining user profiles with virtual trades) efficiently.

6. Context-Awareness in AI Q&A

- **Prompt:** "Answer user questions about their stocks using an AI model."
    - **Human Implementation:**
        - **Context Injection (RAG):** Before sending the user's question ("Should I sell?"), the system fetches the user's *current portfolio* from the database and appends it to the system prompt ("User owns: AAPL @ $150").
        - **Result:** The AI provides specific advice: "Since you bought AAPL at $150 and it's now $170, taking profit might be wise."
    - **AI Agent Implementation:**

        - **Logic:** Simply acted as a proxy, forwarding the raw question to the API.
        - **Result:** The AI responded generically: "I don't know what stocks you own, so I cannot give specific advice."

7. Hedge Analysis Strategy (Math vs. Visuals)

- **Prompt:** "Find stock pairs that move in opposite directions to reduce risk."
    - **Human Implementation:**
        - **Logic:** Implemented the Pearson Correlation formula manually to categorize pairs into actionable groups (Negative, Low, High Correlation).
        - **Visualization:** Integrated Chart.js to render a **Color-Coded Heatmap**, allowing users to visually spot hedging opportunities (Green) vs. risk clusters (Red) instantly.
    - **AI Agent Implementation:**
        - **Logic:** Imported standard Python/Pandas libraries (df.corr()) to calculate the matrix instantly.
        - **Result:** It returned a raw 2D array of numbers. While mathematically accurate, it lacked the **Presentation Layer**—it didn't provide any visual context or specific textual suggestions (e.g., "Buy Stock A to hedge Stock B"), making the data hard for non-experts to interpret.

6.4 Code Quality & Maintainability Analysis

Beyond functional correctness, we evaluated the long-term viability of the codebases.

1. Modularity vs. Monolithic Structure

- **Human Code:** Follows a strict **Model-Controller-Service** separation. Database queries are isolated in the Data Access Layer, business logic is handled in Services, and HTTP handling is performed in Controllers. This makes unit testing individual components (like the Risk Calculator) straightforward.
- **AI Code:** Often generated "Spaghetti Code" where SQL queries, business logic, and HTML response generation were mixed within a single route handler. Modifying one part (e.g., changing the database table name) would require rewriting the entire function.

2. Hardcoding vs. Configuration

- **Human Code:** Uses environment variables (.env) for sensitive data like API keys, database credentials, and adjustable parameters (e.g., risk tolerance thresholds).
- **AI Code:** Frequently **hardcoded** API keys and database connection strings directly into the source code. It also hardcoded values like the "Risk Free Rate" (0.02) inside the Sharpe Ratio function, making it difficult to update without code changes.

6.5 Security Vulnerability Assessment

Security Aspect	Human Implementation	AI Agent Implementation	Vulnerability Level (AI)
SQL Injection	Uses **Parameterized Queries** exclusively.	Used Template Literals (e.g., SELECT ... WHERE name = '${name}') in some instances.	**Critical**
Password Hashing	Uses **bcrypt** (Salt rounds: 10).	Used fast hashing (SHA-256) or MD5 without salt.	**High**
Session Mgmt	Server-side session store with HTTPOnly cookies and secure flags.	Simple client-side tokens or in-memory arrays that vanish on server restart.	**Medium**
Error Exposure	Generic error messages ("Invalid credentials") to prevent user enumeration.	Returned raw database errors (e.g., "Duplicate entry for key 'email'") to the frontend.	**Medium**

6.6 Conclusion & Verdict

The comparative analysis reveals a clear distinction between **coding capability** (syntax generation) and **system engineering** (architecture and reliability):

1. **Code Generation (AI Wins):** The AI Agent is incredibly efficient at writing boilerplate code and standard algorithms. It implemented the **Hedge Analysis** (Correlation Matrix) module faster and with fewer syntax errors than manual coding, proving it is a powerful accelerator for standard tasks.
2. **System Architecture (Human Wins):** The AI struggled significantly with "tacit knowledge"—requirements that are not explicitly stated in a prompt but are essential for professional software. This includes **Security Standards** (choosing bcrypt over md5), **Fault Tolerance** (handling partial batch failures), and **Domain Specifics**

(Annualizing volatility).

Final Verdict: While the AI agent successfully "Passed" the majority of functional tests, its output was essentially a **prototype**. The **Human Implementation** was strictly necessary to transform these prototypes into a secure, robust, and mathematically accurate financial platform. The experiment confirms that while AI can replace the "Typing" of code, it cannot yet replace the "Engineering" of systems.

Contributions

Group member	Responsibility
Yiwei Li	Backend Design and Programming Ai Agent Design and programming Frontend programming
Yanbo Tong	Google Slides Design Program Report (Part 5 - System Evaluation) Frontend Design: Client-Friendly UI Page
Haoyang Guo	Implement the frontend and backend of the virtual portfolio system Using real-time stock API