Netgrain
Team 5 – Design Document

Collin James, Daniel Luo, Kevin Qu, Haiyan Xuan, Nathan Yi, Charles Yuan

Purpose

We are designing a backtesting platform that supports testing trading algorithms against historical data and against granular, high-frequency data generated based on configurable world parameters.

Functional Requirements

Backtesting Engine

1. "As a trader, I would like to be able to write high-frequency trading algorithms so that I can test how they would perform."
2. "As a user, I'd like to test my algorithm against historical data in a .csv file."
3. "As a user, I would like to be able to save custom presents to rerun the same simulation with different trading algorithms to iterate on my trading algorithms."
4. "As a user, I would like the ability to test my trading algorithm on real, historical data."
5. "As a user, I'd like to test trading algorithms against different asset classes (crypto, precious metals, forex, etc.)."
6. "As a user, I want a plug-and-play preset to simulate a bear market (high volatility, high interest rates. etc.)."
7. "As a user, I want a plug-and-play preset to simulate a bull market (high volatility, low interest rates, etc.)."
8. "As a user, I want a plug-and-play present to simulate a sideways trading market (low volatility)."
9. "As a user, I would like to run my algorithms against many different simulations to get an average performance of the algorithm."
10. "As a user, I would like to be able to give the algorithm a set amount of initial capital to better model real world situations."

Analytics Engine

11. "As a user, I would like a table of all the actions my trading algorithm performed time stamped."
12. "As a user, I would like the ability to download all the metrics pertaining to a simulation or test to my personal computer."
13. "As an investor, I want to be able to track quantitative metrics that can inform me how my algorithms are performing, like risk-adjusted return."
14. "As a user, I would like the ability to see snapshots of the algorithms performed and the values it's using at those moments."
15. "As a user, I would like the ability to adjust how frequently algorithm snapshots occur."
16. "As a user, I would like my recorded profits to reflect the real world (taxes, fees. etc)."
17. "As a user, I would like to be able to compare my trading algorithms with previous ones I've developed to determine if mine under or overperforms."

Testing Module

18. “As a developer, I’d want a module to test how well generated data conforms to historical price activity, such as comparing to baseline data generated using geometric Bayesian motion.”

<u>Data Generation</u>

19. “As a trader, I would like to generate new data that my algorithms can test against to see how they would react in unseen market situations.”
20. “As a user, I would like to configure the timeframe of the generated data (test short, medium, and long term correlation).”
21. “As a user, I want generated data to model real-world market data within a tolerance threshold.”
22. “As a user, I would like to be able to input initial market scenarios that will affect the data generated to assess how my algorithm functions in different environments.”
23. “As a user, I would like the ability to temporarily pause data generation to add more control over the events of the simulation."
24. “As a user, I’d want to adjust market conditions to simulate flash crashes.”
25. “As a user, I’d want to adjust market conditions to simulate bubbles.”
26. “As a user, I’d want to adjust market conditions to simulate markets favoring companies of different capitalizations, including blue chip behemoths (e.g., Google, Meta) or small-cap companies (like biotech startups).”
27. “As a user, I’d want to configure positive and negative earnings reports.”
28. “As a user, I’d want to configure interest rate changes (e.g., Fed rate hikes).”
29. “As a user, I’d want to configure volatility in the market for a specific stock.”
30. “As a user, I’d want to configure liquidity offered by market makers in the market for a specific stock.”

<u>Test Script</u>

31. “As a trader, I would like to be able to test multiple trading algorithm scripts at the same time, so that I can see live differences between how the two algorithms perform on the same provided data.”
32. “As a user, I would like the ability to upload my Python scripts directly to the app so I can develop in my personal development environment.”
33. “As a user, I would like the ability to download scripts written on the platform to my personal device.”
34. “As a user, I’d like to have access to a code editor with autocomplete for the functions provided within the trading Python API.”
35. “As an algorithm tester, I would like to be able to interact with the pip python package manager, so that I can install and use any third party package python libraries that I would like to have access to within my trading algorithm scripts.”
36. “As a user, I would like to have the option to write my trading algorithms in Python or in C++ for greater performance control.”
37. “As a user, I’d like to specify which lot of stocks I’d like to sell (if I have multiple) for tax minimization strategies.”
38. “As a trader, I want to be able to buy and sell assets whenever I want by having my orders filled by a high-volume market maker that reflects real-world price action.”

Dashboard

39. “As a user, I’d like to configure my visual dashboard by placing GUI components anywhere I want within a predefined grid.”
40. “As a user, I would like to support different types of charts (line, candlestick, etc.)”
41. “As a user, I would like an interactive chart that shows the prices of the stocks.”
42. “As a user, I want to be able to interact with a live control panel that would let me influence how the market moves in real time.”
43. “As a trader, I would like to be able to see fine grained (over milliseconds) to model the real world stock market.”
44. “As a long term investor, I would like to be able to see how my stocks perform over weeks instead of just days, so that I can assess the long term gains of an investment strategy.”
45. “As a user, I’d like to ‘overwrite’ historical data with generated data in real-time through interactive interfaces.”
46. “As a user, I want a tutorial to guide me through interface components.”
47. “As a user, I’d like to be able to ‘replay’ and ‘overwrite’ simulated data to shape my ideal market situation if I can’t shape it the first try using the real-time actions.”

Social/User Account

48. “As a user, I want a daily market condition that all users can run against and compare results. (similar to chess.com daily puzzles)”
49. “As a user, I would like the ability to create user accounts with an email and password so that I can login and logout easily and persist data across sessions.”
50. “As a user, I would like the ability to share the metrics and results of a simulation with another user via a convenient link or JSON file.”
51. “As a user, I would like the ability to remix (make a copy/change the simulation) off of someone else’s shared simulation.”
52. “As a user, I’d like to see a leaderboard with the best performing algorithms/metrics across the platform.

Appearance

53. “As a user, I would like the ability to toggle between light and dark mode.”
54. “As a user, I want access to accessibility customizations, like font size and limited animations.”

Non-Functional Requirements

Performance

55. Support refresh rate of <100ms (refresh rate is the frequency at which stock updates are broadcast to clients)
56. Design a write-behind buffer capable of storing 500+ data points (orders, trades, etc.) to be asynchronously persisted to the database
57. Data points being passed into the backtesting engine are handled in a multithreaded manner with 50+ threads
58. Support streaming to at least 1,000 connections in real time via WebSockets

Security

59. Uploaded trading algorithm is private and isolated from other others on the website
60. Use HTTPS protocol instead of HTTP so that the user data is secure
61. Python scripts uploaded by the user for trading are sandboxed so that they do not abuse server resources

Appearance

62. Provide visualization charts and graphs capable of handling 500 data points every second
63. Ability to theme the visualization charts on the dashboard by modifying the colors and line widths

Data Structures

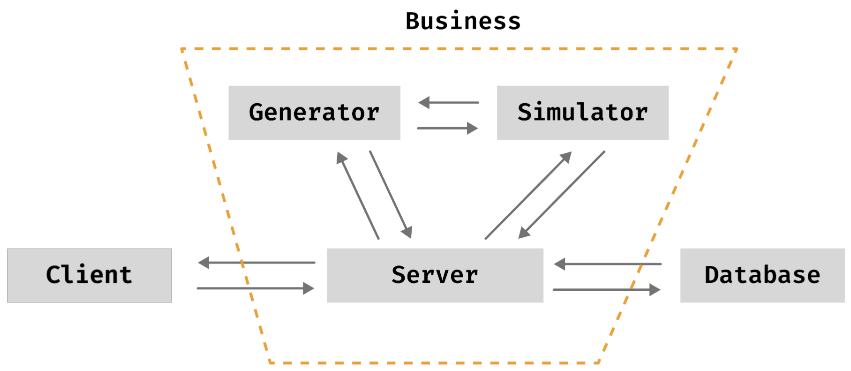
64. Support O(1) order matching using two heaps, one for bids and one for asks
65. Support O(1) order cancellation by storing cancelled limit orders in a separate data structure (instead of traversing through the entire heap)
66. Store all prices and balances as integers (not floating points) to maintain precision; for example, to represent $25.10, store 2510, not 25.10.

Usability

67. Provide a Python API containing 25+ functions that allows the user to access the fine-grained data that our backend has generated in an immediate manner
68. Ability to serialize a simulation run into a binary format or JSON that our simulation engine can read in and rerun at as low as 1% of the original running speed

Design Outline

Model: Three Tier Model



Components

- **Database**: Store information that will not be accessed quickly
    - Store user accounts
    - Leaderboard standings
    - Backlog of daily market conditions

- **Business Logic**: serve user-facing dashboard, generate market data using different modes, simulate algorithm performance against market data
    - Server: Handles and manages request/interactions between all components
        - Authenticates user logins
        - Serves the webpages to the users
        - Streaming selected data for simulated market
        - Receive scripts from users, and sends to simulator
        - Receive configuration of initial market conditions, and sends to generator
    - Data Generation: Generates the virtual stock data
        - Generate downloadable file of initial market conditions
        - Stores the historical data
    - Simulator: Runs the user's trading algorithm on the selected stock data
        - Set up market conditions from file received from server
        - Runs simulations of market data with given scripts from server
- **Web Interface**: A dashboard allowing the user to see and interact with the service
    - Displays graphs of stock prices
    - Shows various metrics on the performance of the user's trading algorithm
    - Shows user-controlled, real-time market event actions

Interactions between individual system components

- Persistence:
    - Market data will be persisted from the market data generator to the database
    - User data (e.g., login information like email and password) will be persisted from the dashboard backend to the database
- Data streaming:
    - Generated data for each supported stock will be streamed to the backtesting engine
    - Generated data for each supported stock will also be streamed directly to the user dashboard via WebSocket
    - Results from the backtesting engine will be streamed to the analytics engine via WebSocket
- User request flow
    - General user requests (e.g., authentication, dashboard GET) will be handled by a standard REST API
    - Simulation events entered by the user are transmitted via WebSocket

Functional Design Issues

1. How often should users be able to run simulations?
    a. **Option 1: Limit to one simulation at a time. After the simulation is done, the user can begin another simulation.**
    b. Option 2: The user has a token bank, which refreshes every day. The user uses the tokens to create simulations. The more complex or data points in the simulation, the more tokens it will cost to create
    c. Option 3: Cap user to a finite, but multiple amount of simulations.

Justification: Simulations will be requested from different users at the same time; therefore, it's safer to limit each user to one simulation to save server resources and enhance user experience. Additionally, by keeping the simulation limit to one per user, it'll make the server design more simpler, since we don't have to worry about race conditions when writing info to the same user.

2. How are we generating data?
    a. Option 1: Take appropriate snippet from historical data and append it to current
    b. Option 2: Train and Deploy an AI model that can look at past patterns and generate simulation as the user requests
    c. Option 3: Using the historical data, we add random noise to create a unique market within a certain tolerance threshold

Justification: Our goal is to provide unique data that hasn't been found in the historical market. Consequently, we have options 2 and 3 to choose from. Option 3 is the best choice here, since we can apply various statistical and random noise onto historical data in a fast manner. Additionally, we can fine tune the level of "noise" there is.

3. How are we streaming the data to achieve high performance?
    a. Option 1: Traditional Request-Response model
    b. Option 2: Server-sent events
    c. Option 3: WebSockets

Justification: Since the goal of our project is to send data in real-time with high throughput, we can have web sockets set up on both the client and the server that transmits smaller data chunks as the simulation goes on. WebSockets are perfect since there's no need for headers for every data packet sent in our three tier model, so there's reduced bandwidth usage.

4. How often are metrics updated to the user?
    a. Option 1: After the simulation ends
    b. Option 2: At a regular interval

Justification: The main selling point of our project is to provide a high-throughput, real-time experience for our users. Therefore, we plan to update the metrics of the simulation in real-time as the simulation runs.

Non-Functional Design Issues

1. How should the user-submitted code be contained?
    a. VM
    b. **Docker**
    c. WebAssembly

Justification: We are going to use Docker because Docker simplifies the process of obtaining dependencies for a user's trading algorithm. Additionally, it is more light-weight than using a virtual machine, while still offering the same level of comparamentalization. Compared to WebAssembly, Docker is easier to use and learn than WebAssembly.

2. How are we going to keep user's code secure
    a. **Will delete user scripts when the simulation is done**
    b. Encrypt the user scripts
    c. Ensure the database is secure

Justification: We will delete user scripts when the simulation is done. We chose this option as opposed to encryption or securing the database because encrypting the scripts will add lots of overhead. Additionally, choosing a secure database will likely come at performance costs and complexity. By not storing the user's data, there will be no opportunities for malicious attackers to obtain user scripts from the database or server. The attacker will not be able to steal scripts from currently running simulations as the simulations will be compartmentalized and unavailable to users.

3. What web service should we use
    a. **AWS**
    b. Digital Ocean
    c. Railway
    d. Heroku

Justification: We want to leverage AWS's ability to scale leveraging its auto scaling feature to run thousands of concurrent simulations all at once without performance degradation. The platform also has CPUs that meet the required power needed to run latency sensitive, high-frequency algorithms in real time.

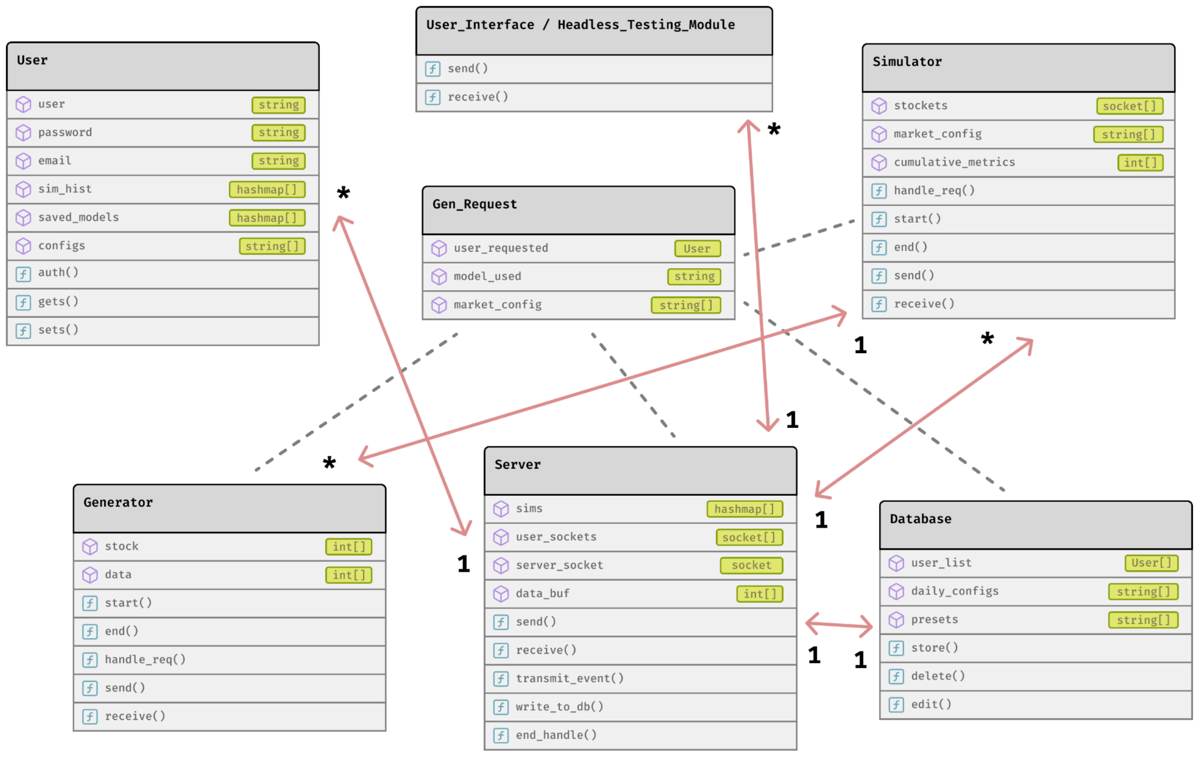
4. What language is the core logic being written in
    a. **C++**
    b. Golang
    c. Pascal
    d. Rust

Justification: C++ is suitable for performance critical code. It also has a large community and a variety of resources (libraries, tutorials, documentations) usable to not only speedup development, but also help to debug and resolve issues within the code.

5. What language/framework are we using to render the graphs
    a. **Use a pre-built JS library like lightweight-charts**
    b. WebGPU

Justification: We will use pre-built JS libraries like lightweight-charts because of better ease of use compared to more low-level implementations like WebGPU. Additionally, lightweight-charts comes with built-in real time streaming updates and have many customizability options, enabling a less intensive learning curve than WebGPU.

Design Details

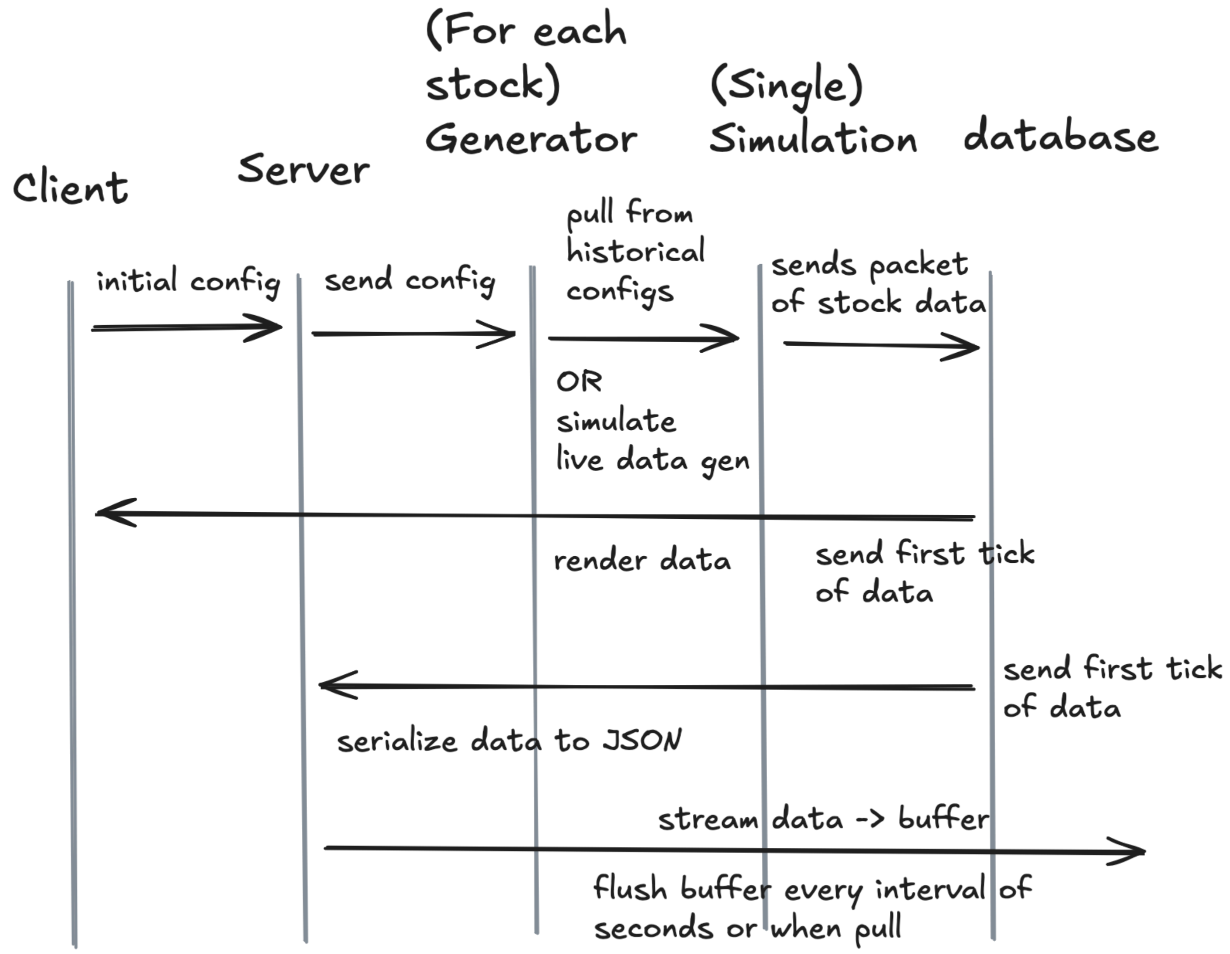


Description of Classes and Interaction between Classes

- **User:**
  - A user is created when someone signs up into the application
  - Each user has a unique username
  - Each user will enter their username, password, and email when signing up for login purposes
  - When logging in, the user will only need to enter username and password
  - Each user will be able to save simulation seeds, as well as have a set amount of most recent simulation seeds saved to our database
  - A user will be able to update their information, as well as update a settings panel and have that information saved and remembered for all following sessions
- **General Request**
  - A general request is created any time which one system wants to interact with another system.
  - A General request contains the user making the request, the market model the request is being sent to, and a config which contains the way they wish to update the market
  - A general request consists of only data which is used by the receiving system to carry out the action indicated within the request.
- **Server**
  - A server is the created when the application is running

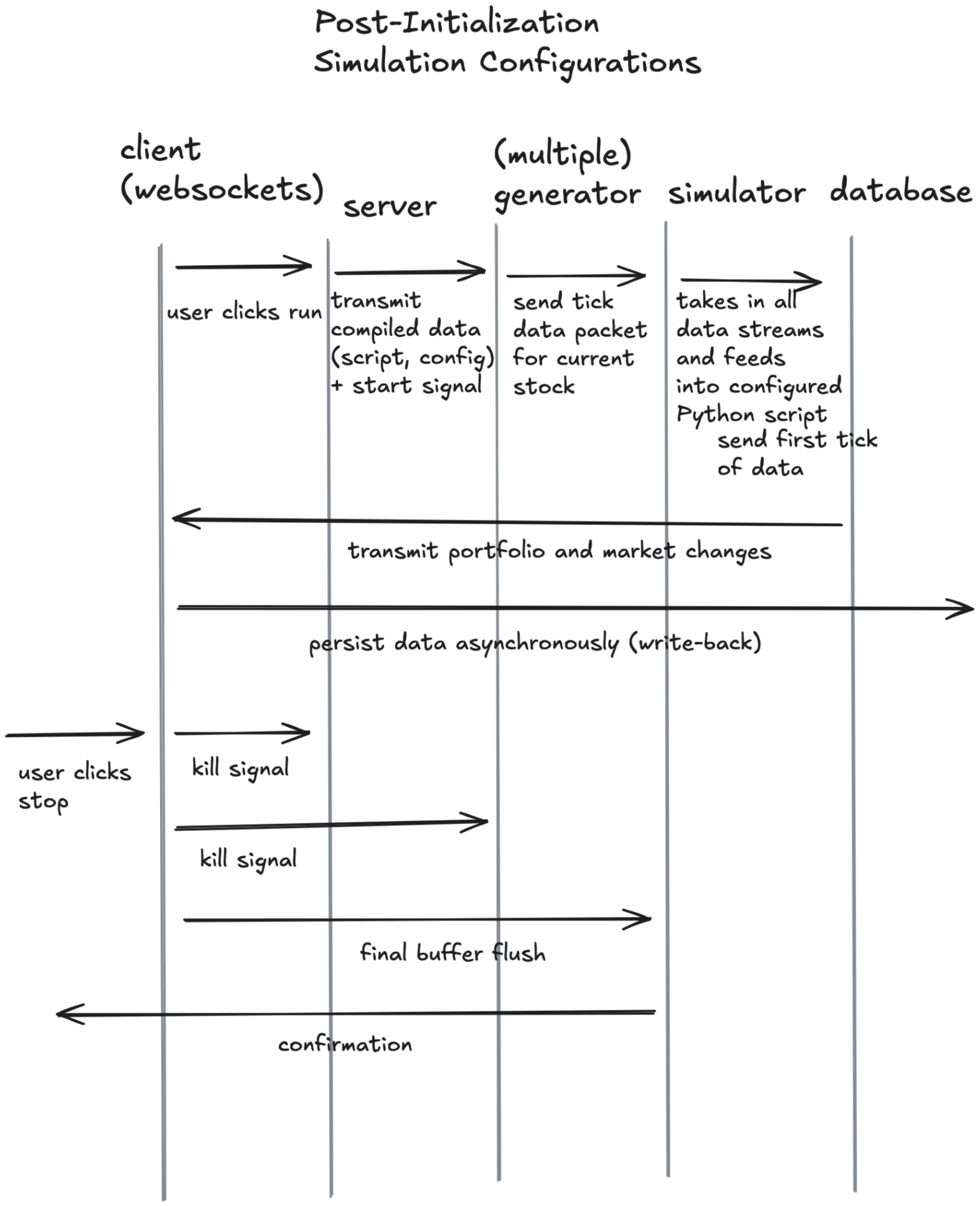
    - A Server contains a list of simulators, current users, a main server socket, and a data buffer consisting of general requests
    - A server is responsible for sending information to and from all of the different components of the application
    - A server will be able to write information of a session back to the central database, to ensure relevant data is persisted between sessions
- **Simulator**
    - The simulator is responsible for taking a user's python script, and data generated from generators, and running a fine grained simulation of a stock market.
    - The simulator has a list of generator sockets, the current market configuration, and a metric analyzer
    - A simulator can start a simulation, end a simulation, and send + receive signals to generators and server,
    - A simulator will take data generated from the various stock generators, and carry out logic present in the user's python script
    - A simulator is responsible for updating relevant market events based on requests sent from server, and updating the generation settings of relevant generators based on the request
- **Generator**
    - A generator is a class designed to serve as a realistic stock simulation for a single given stock
    - A Generator has a stock it is generating, and relevant information related to generating stock prices
    - A generator can both start generating new data, and stop generation
    - A generator can receive stock specific information from a simulator, and adjust generation data based on the request
    - A Generator can stream generated data back to a simulator
- **Database**
    - The database is used to persist user data, and store other data that might need to be accessed by the server or users
    - A database holds a list of all user data, as well as a list of config files that are used for daily market simulations
    - A database can write in new data, update existing data, or remove data
    - A database can fetch data requested by the server

Sequence of Events for Initial Simulation Setup/Run



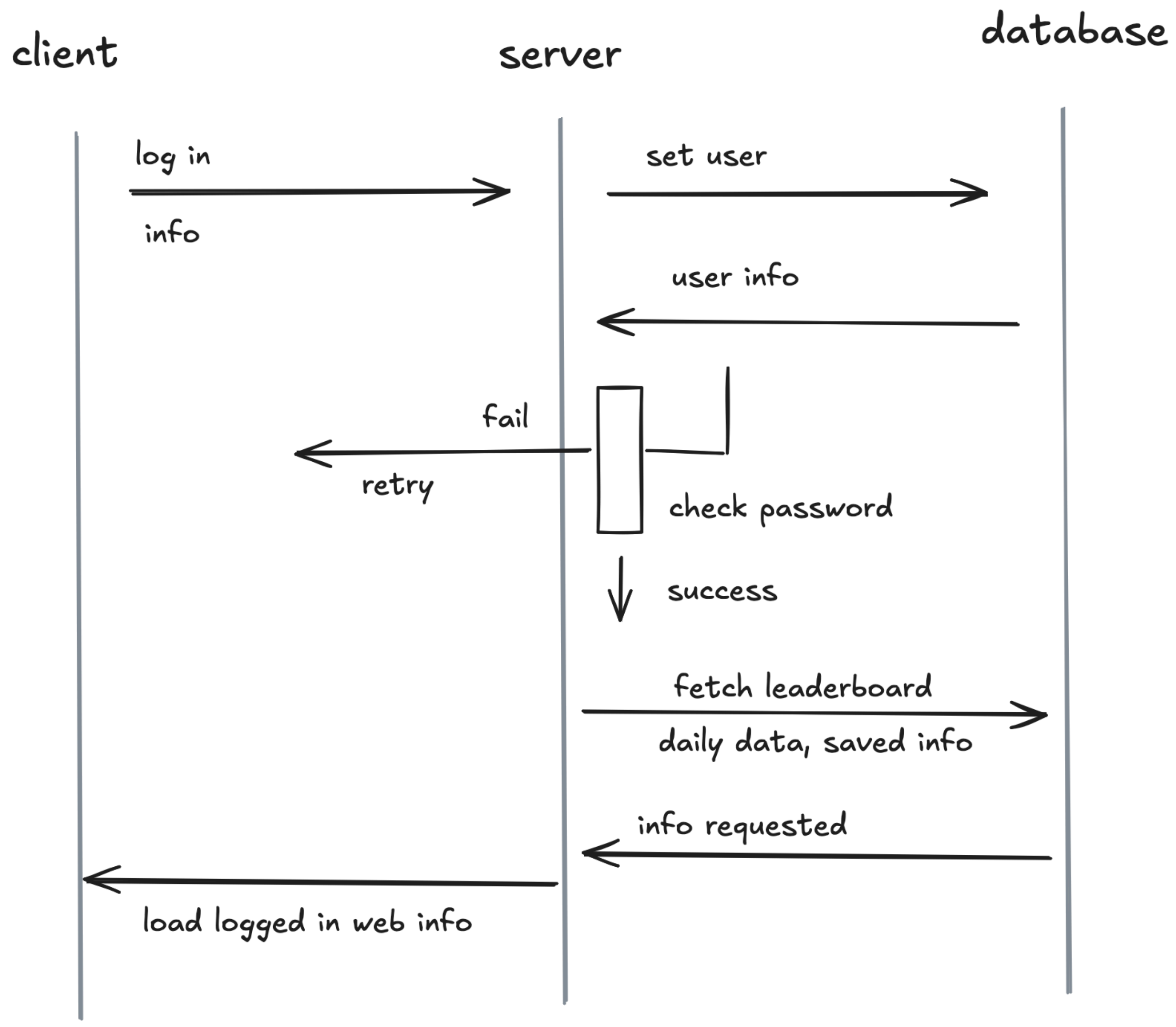
Description: The initial configuration of market conditions (i.e. simulation/generation) is sent from the client to the server. The server fetches the appropriate historical or generated data, and the first tick is returned and rendered on the client-side. Additionally, the data is serialized as a JSON file to initialize a record for future reference for the user. This file will hold important details of the simulation.

Sequence of Events for Post-Initialization Simulation Configurations


Post-Initialization Simulation Configurations
client (websockets)
server
(multiple) generator
simulator
database
user clicks run
transmit compiled data (script, config) + start signal
send tick data packet for current stock
takes in all data streams and feeds into configured Python script
send first tick of data
transmit portfolio and market changes
persist data asynchronously (write-back)
user clicks stop
kill signal
kill signal
final buffer flush
confirmation

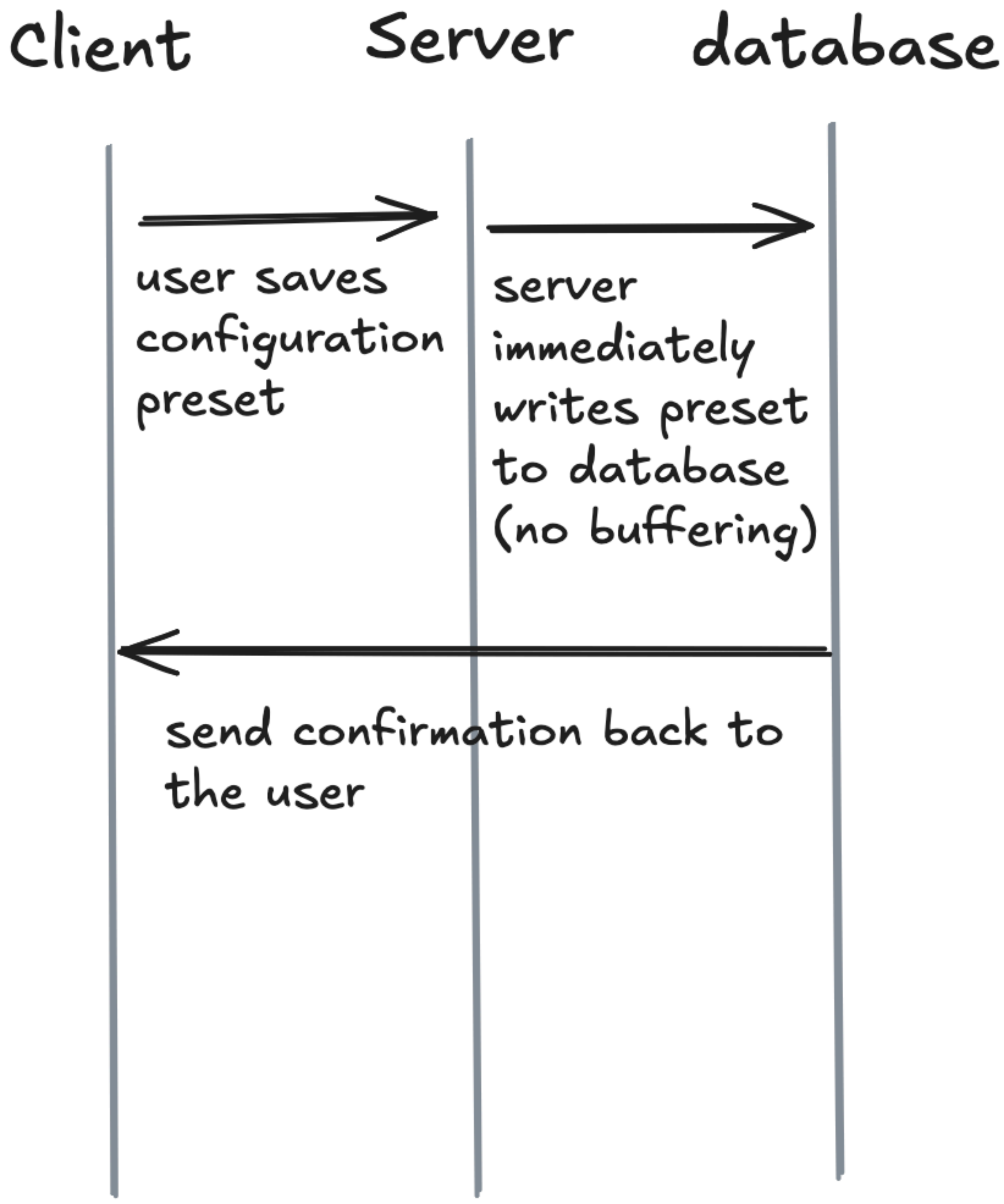
Description: When the user begins a simulation, the trading script/algorithm and simulation configurations are sent to the server. Each stock is instantiated with its own generator. The simulator will then aggregate all of the selected stocks and return a set of cumulative metrics back to the client. The data will asynchronously persist to the database via a write-back buffer implemented in the server. The user will then end the simulation, which will halt any further simulation progress and flush the buffer, ensuring data integrity.

Sequence of Events when User Authenticates



Description: When the user logs, their credentials are checked against the user database. If authentication fails, an error message is returned to the user. Otherwise, the user is successfully authenticated and information like past simulations and leaderboard status are sent to the UI.

Sequence of Events when User Saves a Preset to the Database


Client
Server
database
user saves configuration preset
server immediately writes preset to database (no buffering)
send confirmation back to the user

Description: The user is able to save configuration presets (e.g. base price, liquidity, volatility) for future usage. This information is sent from the client to the database for persistence. The user is notified that their preset was successfully saved.