

LOW-LEVEL ARCHITECTURE AND DATA MODELS

03:AUTONOMOUS TRADING BOT

STUDENT ID	NAME
23100011	Suleman Mahmood
23100197	Ahmed Tahir Shekhani
23100198	Ali Asghar
23100176	Syed Talal Hasan

TABLE OF CONTENTS

1.	2
2.	3
2.1	Architecture Diagram—4
2.2	Architecture Diagram—6
3.	9
4.	10
5.	13
6.	13

1. Introduction

A web application with an autonomous trading bot instance that will trade to generate profitable returns on stocks. The bot will be trained on the PSX data. The bot's decision will be based on the concepts of game theory, mathematical models, financial techniques, and especially artificial intelligence. The primary web app will allow the user to provide the bot's configuration, which includes, target return, risk appetite, and duration of the instance.

With recent advancements in deep learning frameworks and access to faster gpus, training complex models that can predict on time series data has opened new avenues to explore stock market trading. We plan on using models that have a memory component in them, such as LSTM (Long Short Term Memory) to make predictions and trades on the stock market.

The overall objective for the application would be to achieve the return target provided by the analyst while configuring the bot and minimize loss according to the risk factor provided. The potential users of this application would be trade analysts or managers who will use the bot to run its instances according to their requirements. The investors will use it to view their reports of the investments. They can filter the data according to the date ranges as well.

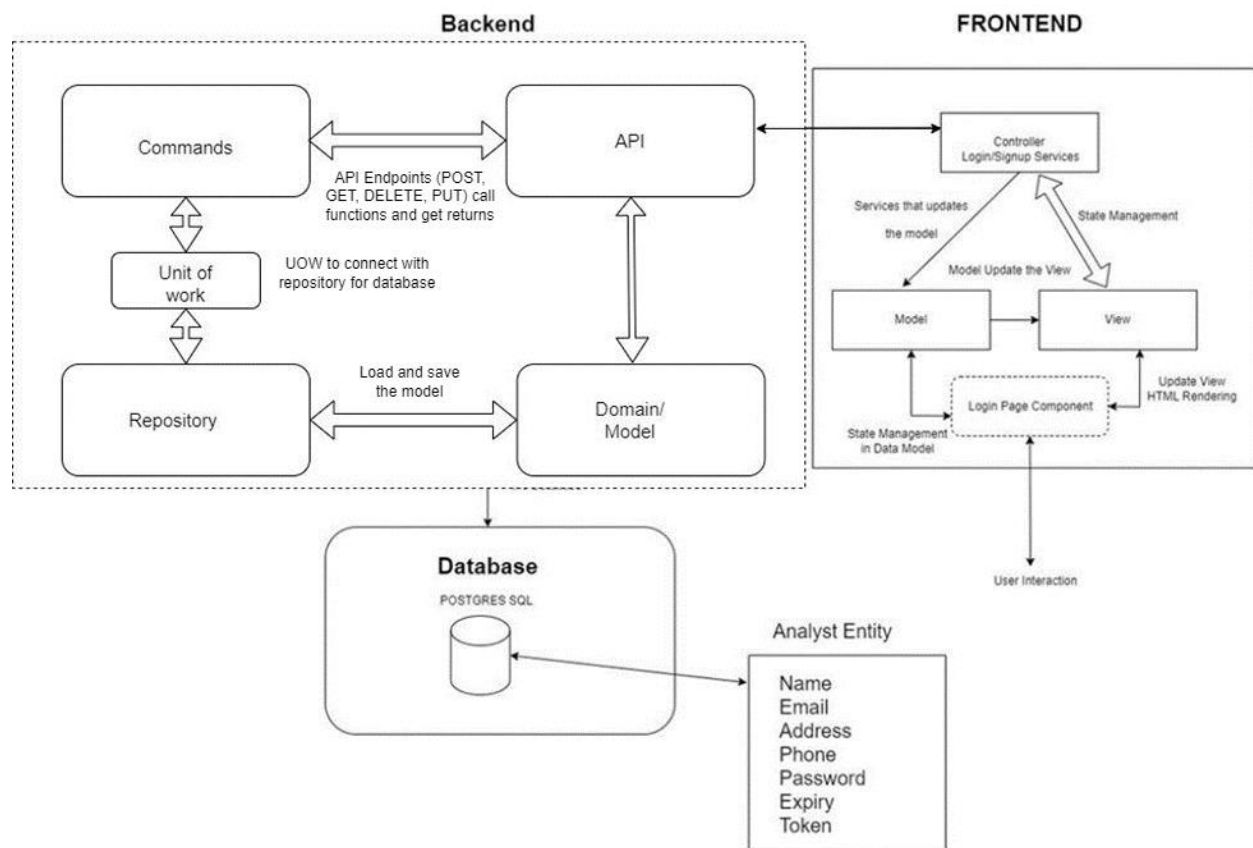
Technical details:

The project's tech stack would be Next.js for frontend web application, Flask for backend server, and PostgreSQL for our persistent storage. The application would follow three-tier architecture with a repository pattern for the persistent layer, models, and command layer for modifying the state.

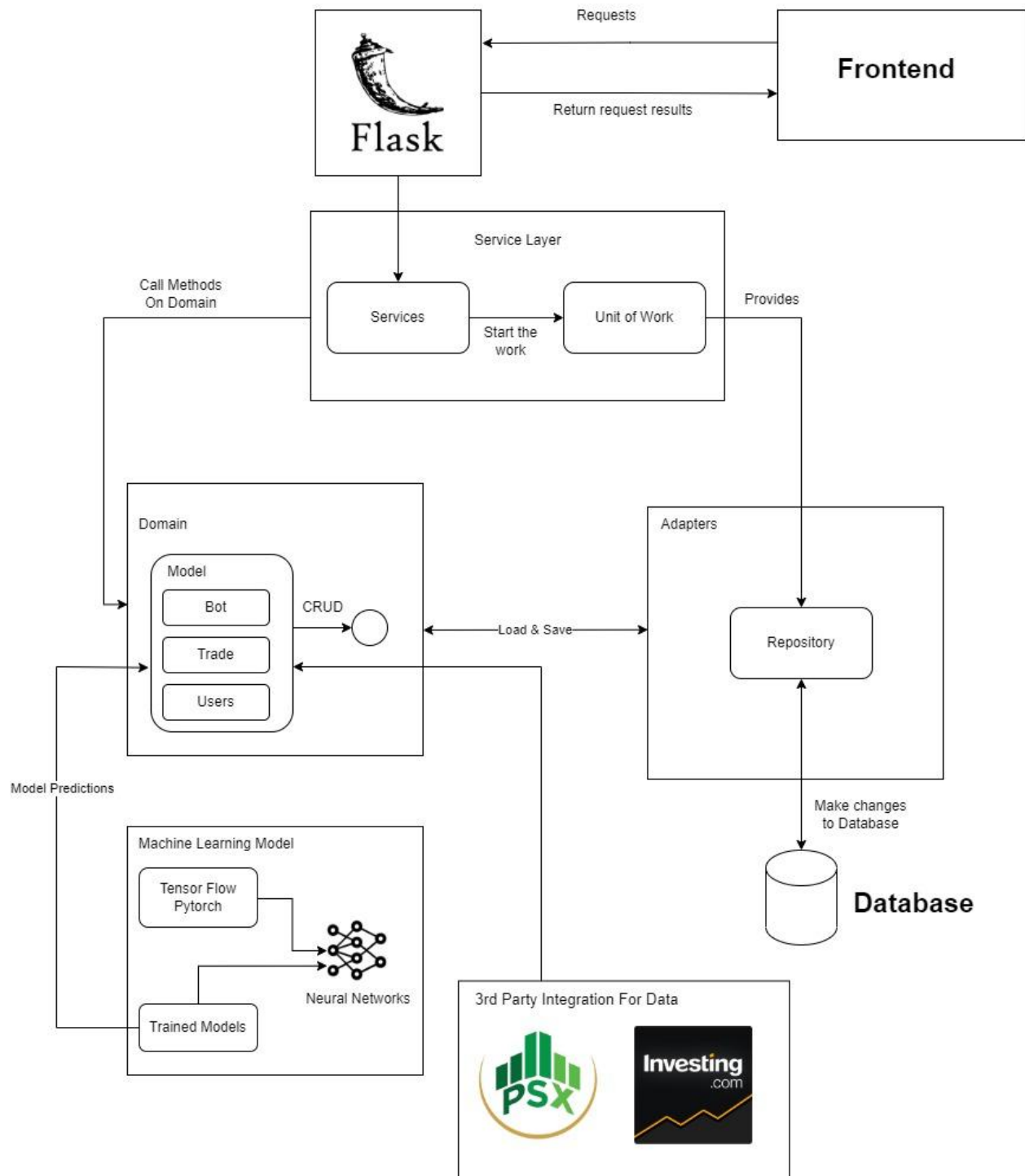
2. System Architecture

2.1 Architecture Diagram—As it is in the prototype code

[It is hard to follow the backend part of the diagram. Are these layers drawn in circles? Follow standard diagram models as are used in Software Engineering.]



Backend Architecture based on Domain Driven Design



User sends a request to the login page component that updates the model and the view. The request is made to the controller that hosts the login/signup service. This request is sent to the flask API. Login page component has Data Model managed in state which renders the view when an update is triggered in data model. For example, in login we have data request of Email and Password along with the analyst or investor role stored in state. As user interacts the view and provide the details, email password and role are shared to model and passed to controller. In services there is loginAuth function which takes the data and make a call to api with axios.post on endpoint /api/v1/analyst_login and waits for the result. This api calls the command with Analyst Unitofwork which attach itself to AnalystRepo. This Repo loads AnalystClass domain with function of Login and pass the database results along with the user provided email and password. Domain function of “login” use hash to compare password and email and return true if success. The model is updated on backend and data gets return from api. On retrieving success message the controlled renders dashboard view and the user gets the access dashboard functionality now.

2.2 Architecture Diagram—As it should-be

[You need to rephrase this whole section. Your arguments must be clear and easily readable. Moreover, you should also make it clear how this architecture is different from your system architecture in the prototype.]

In a couple of paragraphs, discuss how your updated architecture will help in achieving design goals listed above.

The architecture we followed in the prototype is the MVC, which will help us maintaining our data models and data states at the Model part while all the interaction and html rendering is based at the view component while all the API handling and promises are controlled by the controller itself. This architecture follows separation of concern and keep our code as much independent as possible so change in one component doesn't affect the other. For example, if we create the bot instance and define its configuration parameter and make any changes to the API endpoints or API call section particularly the controller module then our codebase of view would not be affected.

As the MVVM model will allow for the separation of components in each layer, we can test them separately. This allows for the code base to be easily tested if new components are added and integrated.

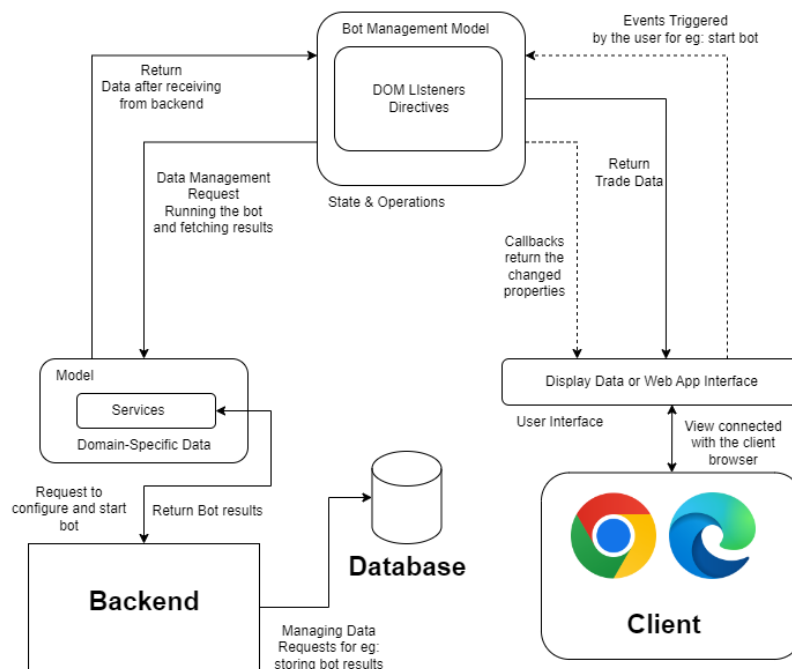
It is reusable as if we change the underlying View Model to just a controller the components of the view and model don't have to be changed. We can reuse them in the updated architecture

as needed. It also allows these components to be incorporated in the extension of the code if we add new features.

As the model is separate the data management and the front end logic can be extended here. The view model can manage multiple instances of the model along with event listeners which makes it scalable as we can add other instances on the existing ones.

As the model view is the bridge between the model and the view, each class has different concerns that it can call. The bot class cannot interact with the login/signup logic nor vice versa. These implementations are kept hidden from separate concerns in the MVVM architecture.

Frontend - MVVM Architecture



how this architecture is different from your system architecture in the prototype:

Our backend architecture is almost same, but our main changes in the architecture is the frontend side. In our main application architecture we intend to work on MVVM architecture but for Prototype keeping the resource constraints in mind we shifted to MVC architecture. Instead of event handling as in MVVM and particularly ViewModel Module we are using controller which interact with the backend section of the application. So for eg, our creation of bot instance APIs and other such RESTFUL APIs are handled by controller part of the architecture.

User sends a request to the login page component that updates the model and the view. The request is made to the controller that hosts the login/signup service. This request is sent to the flask API. Login page component has Data Model managed in state which renders the view when an update is triggered in data model. For example, in login we have data request of Email and Password along with the analyst or investor role stored in state. As user interacts the view and provide the details, email password and role are shared to model and viewModel. The ViewModel will use the services and call api with endpoint /api/v1/analyst_login and waits for the result. This api calls the command with Analyst Unitofwork which attach itself to AnalystRepo. This Repo loads AnalystClass domain with function of Login and pass the database results along with the user provided email and password. Domain function of "login" use hash to compare password and email and return true if success.

3. Data Models

[It is hard to understand the purpose of each entity. Description should be clear. Moreover, you should also model data from external sources such as stock exchange. How are authorization and authentication being managed?]

Entities:

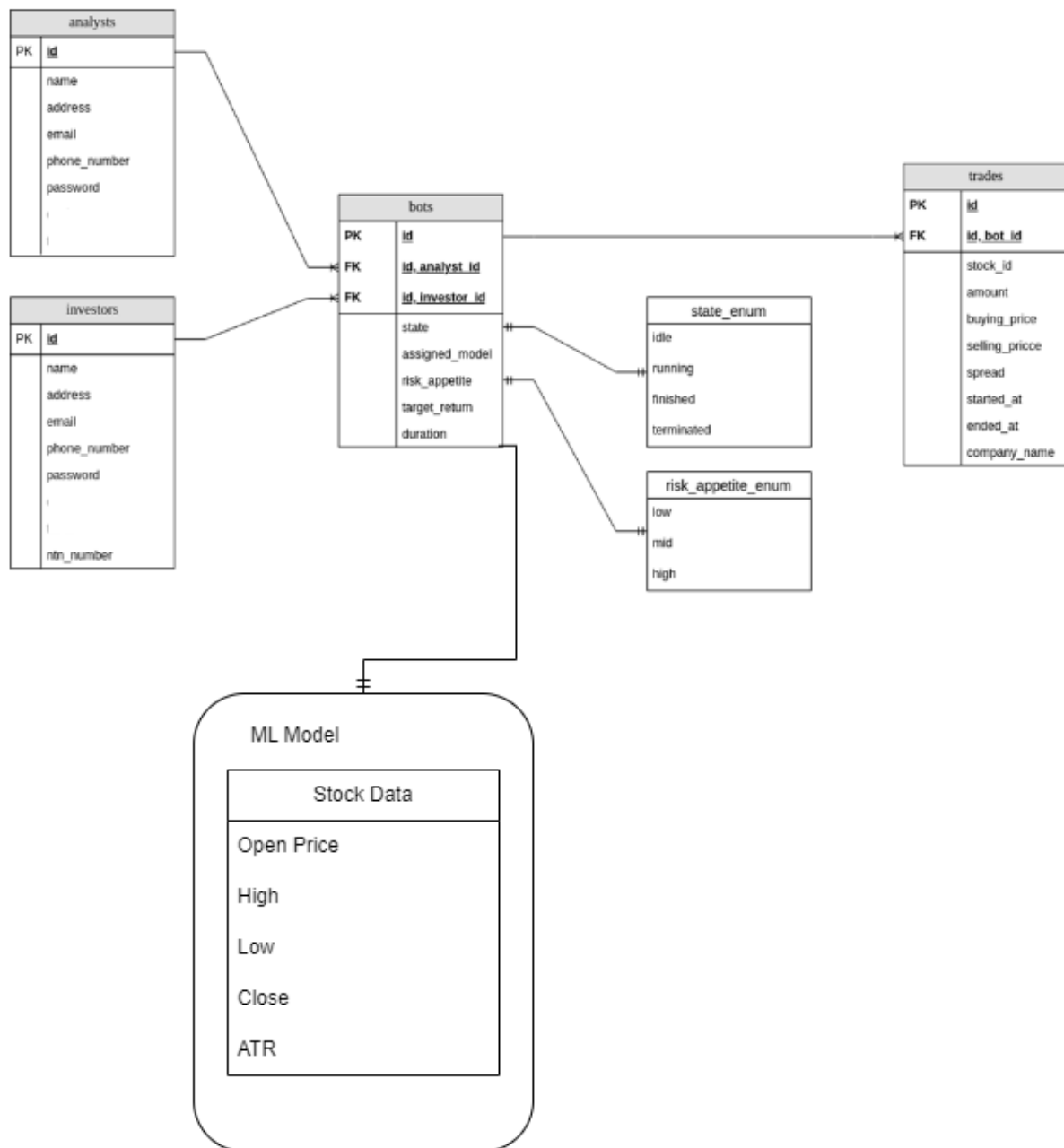
Analyst: Analysts will register investors, configure and start the execution of the bot, analyze its statistics and assign investors to their bots. Analyst are the main user of the application who will register the investor and get the credentials in return that can be shared to investors in-person. Analyst table is representing a separate entity which means we could change the personal data of analyst won't affect the performance or functionality of the analyst. They can create Bot instances which connects them with the investor they create a bot for.

Investors: Investors will use the bot for investment purposes and can view a summary report of the performance of the bot for a specific range of time period (monthly, quarterly, half-yearly, yearly, or other combinations of days or months). Investors use the credentials provided by the analyst and they can see the bot's performance. Their data is independent of any other entity. The created Bot on their behalf by the analyst helps them connecting with the analyst so they can see their portfolio build by the individual analyst of the company.

Bot: The bot will be trained on the PSX data. The bot's decision will be based on the concepts of game theory, mathematical models, financial techniques, and especially artificial intelligence. The primary web app will allow the user to provide the bot's configuration, which includes, target return, risk appetite, and duration of the instance. The ML model based on Tensorflow libraries is used to return the prediction of stock prices of 14 days which help the bot to make decision. Bot has an investor id and an analyst id so it's a relational entity to connect analyst and an investor. The ML model is trained using the data from stock exchange (PSX) to make future predictions.

How are authorization and authentication being managed?

For authentication, only login and password are required from the user where the role will be automatically fetched. It will first compare the email in analyst table and if couldn't find then investor, then after successful email search it will compare the hash of input password and password stored in database. After successful attempt JWT is created which is stored in localStorage with an expiry of 1 hour. On every call over dashboard to perform any authorized function, JWT needs to be passed in Authorization header of a RestAPI call.



4. Tools and Technologies

○ Frontend

Language

- Typescript 4.8.4
- ESLint 8.25.0

JavaScript library

- Next.js 17.0.2
- React.js 18.2.0

CSS Framework

- Tailwind CSS 3.1.8
- DaisyUI 2.31.0

API library

- Axios 1.1.2

Deployment

- Vercel

○ Backend

Language

- Python 3.10.7
- PostgreSQL 15

Persistent storage

- PostgreSQL 15

Server

- Flask 2.2.2

Deployment

- Google Cloud's App Engine flexible

- Deep Learning Models:

Language

- Python 3.10.7

Frame work

- Pytorch and Tensorflow 1.12.1

Tool

- Google Colab

5. Who Did What?



Name of the Team Member	Tasks done
Suleman Mahmood	ER Diagram
Ahmed Tahir Shekhani	Architecture Diagram(2.1,2.2), Introduction, Description
Ali Asghar	Tools and Technologies, Data Models description
Syed Talal Hasan	Updated Architecture description

6. Review checklist

Before submission of this deliverable, the team must perform an internal review. Each team member will review one or more sections of the deliverable.



Section Title	Reviewer Name(s)
Architecture Diagram(2.2)	Suleman Mahmood
ER Diagram	Ahmed Tahir Shekhani
Architecture Diagram(2.1)	Ali Asghar
Tools and Technologies	Syed Talal Hasan