International Scientific Journal of Engineering and Management (ISJEM) ISSN: 2583-6129
Volume: 04 Issue: 05 | May – 2025 DOI: 10.55041/ISJEM03906
An International Scholarly || Multidisciplinary || Open Access || Indexing in all major Database & Metadata

AI-Powered NSE Stock Paper Trading Web Application

Tarun Jain[1], Nitin Kumar[2], Vidit Nagar[3], Piyush Gupta[4]
Department of Computer Science and Engineering (Artificial Intelligence and Machine Learning)
Inderprastha Engineering College, Ghaziabad, India

1. Introduction

Paper trading — simulating trades without real money — is a crucial tool for aspiring traders to practice strategies in a risk-free environment. It allows investors to buy and sell securities without financial risk, helping them learn market mechanics and test strategies before committing capitalIn today's markets, artificial intelligence (AI) has emerged as a game-changer in finance, revolutionizing how investors analyze and predict stock movements . Machine learning models can uncover patterns in vast historical data and avoid human biases, often improving prediction accuracy and consistency. Notably, AI-powered funds and algorithms have outperformed traditional methods, indicating that AI can be the decisive factor between seizing opportunities or missing them in volatile markets . AI techniques like price prediction models and sentiment analysis of news are increasingly used to enhance trading decisions.

This paper presents a web-based stock paper trading application for the National Stock Exchange (NSE) of India that leverages AI for educational and strategy-testing purposes. The platform provides users real- time NSE market data and allows them to execute simulated trades (no real money), while an integrated machine learning module predicts stock prices and an NLP module analyzes market sentiment. Such an application offers a sandbox for beginners and experienced traders alike to learn, practice, and validate trading strategies with guidance from AI insights. The rest of this paper details the system's design and components: an overview of features, the technical stack employed, data processing and modeling methodology, system architecture (with diagrams), example use cases and benefits, performance results, limitations, and future enhancements.

2. System Overview

The AI-powered paper trading app provides a rich feature set to emulate a trading experience and augment it with predictive analytics:

- Simulated Trading Environment: Users can create a virtual trading portfolio and are allotted a certain amount of virtual cash to start (e.g. ₹1,00,000). They can place buy/sell orders on NSE stocks through the app, which executes these trades at the real-time market prices but in a simulated ledger. The account balance and holdings update based on profit/loss, just as with a real brokerage account. This allows users to practice trading without risking real money — they can experiment with different trading techniques and learn by trial and error . All trades, positions, and P/L are tracked so that users can review their strategy performance over time.

- Real-Time Market Data Feed: The application provides live price quotes and charts for NSE stocks. It continuously retrieves up-to-date market data (current price, bid/ask, daily high/low, etc.) and displays it to the user with minimal delay. This real-time feed ensures the paper trading experience is realistic — users make decisions on the same data they would have in actual trading. For instance, if a stock's price is ₹250 at 10:30 AM, the platform will reflect that price in real-time and allow a trade at that price. The backend achieves this by periodically fetching data from an online API or website (e.g. official NSE APIs or Yahoo Finance) and updating the front-end view. Live charts can also be provided, updating with the latest price ticks. This feature familiarizes users with watching market movements and responding to live price changes.

- AI-Based Price Prediction: A core feature of the app is its machine learning price prediction module. The system uses a trained ML model to forecast future stock price movement — for example, predicting the next day's closing price or the short-term trend (up or down) for a given stock. When a user selects a stock, the app displays not only the current data but also the model's prediction (with appropriate disclaimers that it's an estimate). For instance, the app might indicate

"Model predicts +2.5% price increase for tomorrow" based on historical patterns. This helps users learn how to incorporate model forecasts into their strategy. The prediction is generated using historical price data and technical indicators fed into a machine learning algorithm (described in Section 4). By comparing predictions with actual outcomes in their paper trades, users can gauge the model's accuracy and adjust their trust in such AI tools accordingly.

- Market Sentiment Analysis: In addition to numeric predictions, the application provides a sentiment analysis indicator for each stock. It analyzes recent news headlines, articles, or social media posts about the company/market to assess the sentiment as positive, negative, or neutral. For example, if major news about a company is very favorable, the sentiment indicator might show "Bullish" or a high positive score. This gives users insight into the market mood beyond just price

data. The app might display a summary like "News sentiment: Negative ⚠" if many recent news articles about the stock are pessimistic. This feature is powered by Natural Language Processing (NLP) techniques that parse news text (see Section 4). Sentiment analysis provides context to price movements — for instance, a price drop accompanied by negative news sentiment could confirm a fundamental reason for the decline. Integrating sentiment aims to teach users how qualitative factors like news can impact trading decisions. Indeed, sentiment analysis has become essential for algorithmic trading success, as it can enhance predictive capabilities by quantifying market psychology .

Overall, the system serves as a learning platform. Users can practice trading in real market conditions (via live data) while receiving AI-driven guidance (predictions and sentiment) to inform their decisions. They can test strategies — for example, "Does buying when the model is bullish and sentiment is positive yield good results?" — and iterate without financial risk. This combination of paper trading with AI feedback helps users refine their strategies and gain confidence before they transition to real trading. In summary, the app's features — real-time data, simulated trading, ML price forecasts, and sentiment insights — together create an interactive environment for strategy development and financial education.

3. Technical Stack

The application is built with a combination of technologies across web development and machine learning, chosen to ensure a responsive user experience and robust backend processing. Table 1 summarizes the major components:

- Backend – Java Spring Boot: The core application server is implemented in Java using the Spring Boot framework. Spring Boot was chosen for its rapid setup of production-ready web services and its popularity in building microservices . It provides an embedded web server and a robust MVC

structure for handling HTTP requests (for pages and APIs). The backend uses Spring's REST controllers to expose endpoints for client interactions (e.g. retrieving stock data, executing a paper trade order). Spring Boot's reliability and scalability make it ideal for handling multiple user sessions and integrating with other services. Java also offers strong type safety and performance for the core logic. In this project, Spring Boot acts as the coordinator between the front-end and the Python ML components, as well as managing user sessions and trade simulation logic.

- HTTP Client (OkHttp3) and Web Scraping (jsoup): The Java backend fetches external data (stock prices, news) using HTTP requests. It utilizes OkHttp3, an efficient HTTP client library for Java, to call external REST APIs or download HTML pages. OkHttp is a modern HTTP/2 client known for its performance and reliability (connection pooling, transparent retries, and HTTP/2 multiplexing support) . For example, to update a stock's price, the backend might use OkHttp to call a public

NSE quote API or scrape a webpage. When HTML needs to be parsed (such as scraping news headlines from a website), the project uses jsoup — a Java HTML parser library. Jsoup provides a convenient API to fetch and parse HTML, and to extract specific data using DOM traversal or CSS selectors . In our app, jsoup is used to scrape relevant webpages (like financial news sites or the

NSE's own website) to gather unstructured data (e.g. latest news titles for a stock) which are then passed to the sentiment analysis module. Together, OkHttp3 and jsoup enable the backend to reliably retrieve and process live data from the web in real-time.

- Frontend – HTML, CSS, JavaScript, Bootstrap: The user interface is built as a web client using standard HTML/CSS/JS technologies, ensuring that it runs in any modern browser. The design is made responsive and professional with Bootstrap, a popular front-end framework for layout and styling. Bootstrap provides a pre-built grid system and components, allowing a mobile-first, responsive UI with minimal custom CSS. It is one of the most widely used HTML/CSS/JS libraries for developing responsive web interfaces . Using Bootstrap, the app's pages (like the dashboard,

trading screen, etc.) adapt to different screen sizes and have a consistent, clean look without extensive CSS coding.

JavaScript is used on the front-end to handle dynamic behavior — for example, updating the displayed stock prices periodically (via AJAX calls to the backend), or showing interactive charts. The front-end communicates with the Spring Boot backend via HTTP requests (for example, form submissions for trades or AJAX GET requests for refreshing data). This separation of front-end and back-end concerns makes the app maintainable and scalable. Users interact with the app through a web browser, and all business logic runs on the server or ML service side, keeping the client lightweight.

- Machine Learning Components – Python (FastAPI & Flask) and Datafeed: The predictive analytics features are implemented in Python, leveraging its rich ecosystem for machine learning and data science. The project uses a Python ML service that runs separately from the Java server. Initially, a simple Flask app was used as a proof-of-concept to expose the model via an API, and later a switch was made to FastAPI for better performance and concurrency. FastAPI is a modern, high- performance web framework for building APIs in Python — it can handle many requests concurrently without blocking, thanks to its asynchronous support . This is important for a trading app, as

multiple users might request predictions simultaneously (especially during market hours) and we need low-latency responses. The Python ML service uses FastAPI to define endpoints (e.g. and the Java backend can call. Inside this service, the machine learning model (for price prediction) and NLP model (for sentiment) are loaded. The service is responsible for receiving data (such as a stock ticker, or news text) and returning computed predictions.

For data, the Python side relies on an open-source library called tvDatafeed to obtain historical market data. TvDatafeed is a Python module that connects to the TradingView platform and downloads historical price data for a given symbol and exchange . It supports NSE symbols, allowing up to 5000 bars of historical data per request. In our application, tvDatafeed is used during model training (to gather historical OHLC price series for each stock) and can also be used on-demand to fetch recent data for predictions. By using TradingView's data via tvDatafeed, we leverage a reliable source of historical market data without directly scraping NSE's site. Python's data libraries (Pandas, NumPy) are used to process this data into features for the ML model. For the machine learning algorithm, we utilized libraries such as scikit-learn and TensorFlow/Keras for implementing the predictive model (see Section 4 for details). The NLP sentiment analysis leverages libraries like NLTK or spaCy for text processing, and possibly a pretrained sentiment model (e.g. a FIN-BERT model or a simpler sentiment lexicon approach) to classify text. All these Python-side computations are exposed through the FastAPI endpoints.

- Build & Deployment – Maven and Docker: The development workflow employs Apache Maven for building the Java Spring Boot project. Maven is a powerful build automation and dependency management tool for Java that standardizes project structure and eases managing external libraries

. We use Maven to compile the code, run tests, and package the Spring Boot application into an executable JAR. Maven also handles fetching all required dependencies (Spring libraries, OkHttp3,
jsoup, etc.) via its declarative POM file, ensuring consistent builds across environments. For deployment, the application uses Docker containerization. Each major component is containerized into a Docker image — e.g., one image for the Spring Boot backend (with Java and the compiled JAR), and another image for the Python ML service (with the necessary Python environment and model files). Docker allows packaging the code along with its runtime environment and libraries into a lightweight container that can run on any system with Docker installed. This guarantees portability and consistency: the same container can run on a developer's machine or on a server, yielding identical behavior . We define Dockerfiles to specify how to set up each container (base image, copying code, installing dependencies, etc.). For example, the Python service's Dockerfile starts from a Python 3.9 image, installs required libraries (FastAPI, scikit-learn, etc.), copies the model pickle files, and launches the API. The containers communicate over a network (Docker Compose can be used to orchestrate multi-container setup). Containerization simplifies deployment and scaling — we could run multiple instances of the ML container behind a load balancer if needed, or deploy the whole stack to cloud platforms easily. It also encapsulates the environment differences (Java vs Python) cleanly. In summary, Maven handles the build process and Docker handles the deployment, making the development lifecycle efficient and the system deployment reproducible.

Table 1. *Summary of the technical stack and roles.*

Layer	Technology	Role in System
Backend	Java 17 + Spring Boot	Core web application server; exposes REST API for frontend, handles user sessions, executes trade simulation logic [8] .
HTTP & Data	OkHttp3 (Java HTTP client)	Fetches external data (price APIs, web pages) efficiently (HTTP/2, pooling) [9] .
	jsoup (Java HTML parser)	Scrapes and parses HTML from web (e.g. news sites) to extract data [10] .
Frontend	HTML5, CSS3, JavaScript	User interface – web pages and interactive elements in browser.
	Bootstrap 5 (CSS framework)	Responsive design and layout for a professional look [11] .
ML Service	Python 3 + FastAPI	Hosts machine learning models behind REST endpoints (e.g. prediction, sentiment); chosen for high-performance async requests [12] .
	Flask (Python microframework)	(Used in early development/prototyping for quick API setup; later replaced by FastAPI for production).
	scikit-learn, Keras, etc.	Machine learning libraries to implement and train prediction models (e.g. regression, neural network).
	NLP libraries (NLTK, spaCy)	Natural language processing for sentiment analysis on text data.
	tvDatafeed (StreamAlpha)	Data access library to download TradingView historical price data for NSE symbols [13] .
Database	H2 (in-memory) or MySQL (opt)	*Optional:* To persist user accounts and trade history. *(For this project, an in-memory store was used for simplicity).*
Build	Apache Maven	Java build automation and dependency management for reproducible builds [14] .
Deployment	Docker	Containerization of backend and ML service for portable deployment across environments [15] .
	Docker Compose	Orchestration of multi-container setup (for local testing and linking services).

4. Methodology

This section describes how the system works under the hood – from data collection and model training to generating predictions and integrating them into the web app.

4.1 Data Collection and Preprocessing: The first step was to gather historical and real-time data required for model training and live predictions. For each target stock (NSE listed company or index) that the app will support, we collected historical price data using the tvDatafeed Python library. tvDatafeed interfaced with TradingView to download historical OHLC (Open, High, Low, Close) data. We fetched daily price bars for the past several years for each stock to have a substantial training dataset (TradingView allows up to 5000 data points per request , so for longer history we made multiple requests or supplemented with other data if needed). This data was loaded into Pandas data frames for preprocessing. We created additional features from the raw price series, including technical indicators that are often predictive in trading. For example, we calculated moving averages (e.g. 20-day SMA, 50-day SMA), volatility measures (e.g. standard deviation over last 10 days), momentum oscillators (like RSI — Relative Strength Index, and MACD), etc., for each day. These technical indicators were chosen based on domain knowledge and prior research, as they can help the ML model detect trends and patterns beyond raw prices . The inclusion of such features has been shown to
improve model accuracy; for instance, a study employing indicators like SMA, MACD, and RSI with an LSTM model achieved high prediction accuracy (~93% on trend prediction) . We also labeled the data for training the model — depending on the prediction task, this could be the next-day price (regression) or a binary label whether the price will rise or fall (classification). The data was then split into training and testing sets (typically 80/20 split by time, ensuring the model is evaluated on "future" data it hasn't seen).

In addition to price data, we compiled a corpus of textual data for sentiment analysis. This involved scraping news articles and headlines related to the stocks in our dataset. Using jsoup in the Java backend (or Python requests in some cases), we pulled recent news headlines from financial news websites and Twitter feeds for each company. Each headline was labeled (if possible) with the date and whether the stock moved up or down after the news (for a rough sentiment correlation). However, for the sentiment model, we primarily needed texts and an indication of their sentiment. We leveraged an existing sentiment- labeled dataset for financial news (when available) to train or fine-tune our sentiment classifier. In cases where using a pre-trained model was more feasible, we opted for a pre-trained language model (like FinBERT or a SentimentIntensityAnalyzer) that is known to perform well on finance-related text. All text data was cleaned (removing HTML, converting to lowercase, etc.) and tokenized. Stop-words were removed and stemming applied as needed to normalize the text. This prepared text dataset was then used to train the NLP model for sentiment (or we evaluated the pre-trained model on it to ensure it was adequate).

4.2 Machine Learning Model Training: With processed data in hand, we trained the price prediction model. We experimented with a few modeling approaches: (a) a classical regression model (such as linear regression or random forest) using technical indicators as features to predict the next day's percentage return, and (b) a recurrent neural network (specifically an LSTM – Long Short-Term Memory network) to capture time-series patterns. The LSTM model was ultimately chosen for its superior ability to learn sequential dependencies in stock data. We set up a multi-layer LSTM neural network in Keras, feeding it sequences of past days (e.g. 60 days window) of prices and technical indicators, and training it to predict the next day's closing price. Training was done on the historical data, using the first 80% of the timeline for training and the last 20% for validation/testing. The model's hyperparameters (like number of LSTM units, learning rate, epochs) were tuned to avoid overfitting. We employed early stopping during training to ensure the model generalizes well. The result was a predictive model that, given recent data for a stock, can output a predicted future price or price trend. For instance, the model might learn that if a stock has had a steady uptrend with increasing volume and positive news sentiment, it's likely to continue rising the next day. Model performance was evaluated on the test set: we measured metrics such as mean absolute percentage error (MAPE) for price prediction, or accuracy for direction prediction. We found that the LSTM model could predict the direction of next-day movement with about 70–75% accuracy on average for our test stocks, while the regression models performed a bit lower, so we retained the LSTM for deployment. This accuracy is reasonably high (by comparison, some studies report around 60–65% for daily direction, though some claim higher with specialized features). We acknowledge that these models are far from perfect predictors of market moves, but they provide a probabilistic edge that can be informative in a learning scenario.

For the sentiment analysis, we trained a text classification model to classify news sentiment as Positive, Negative, or Neutral regarding the stock. We used a labeled dataset of financial news (where each news item was labeled by humans or a proxy as having positive or negative implications). A simple approach using a Naive Bayes classifier with a bag-of-words feature representation was initially tried and achieved around 80% accuracy in identifying sentiment polarity. However, we ultimately integrated a pre-trained FinBERT model (a BERT variant fine-tuned on financial text sentiment) which provided more nuanced

understanding of text. This model, when given a news headline or tweet, outputs a sentiment score or category. We tested this on recent headlines and found it aligns with human judgment in most cases (e.g., headlines containing words like "beats expectations" came out positive, whereas ones with "scandal" or "losses widen" came out negative). The sentiment model process involves typical NLP steps: tokenizing the text, feeding into the transformer model, and getting a sentiment classification. For efficiency, we might not fine-tune the transformer on our data due to resource constraints, but even out-of-the-box it performed well for our needs. The sentiment analysis operation is relatively fast (on the order of milliseconds per piece of text using a transformer with GPU or a smaller model on CPU).

After training, the models were deployed as follows: The final LSTM model is saved (serialized) using Keras (an H5 or SavedModel format), and the sentiment model is also saved or loaded (if using a pre-trained one from HuggingFace, for example). These model files are included in the Python service container. The FastAPI service loads these models into memory at startup. This way, predictions can be served quickly without re- loading models on each request.

4.3 Integration and Prediction Workflow: With the trained models ready, we integrated them into the web application. The Python ML microservice exposes two main endpoints: and The Spring Boot backend communicates with these endpoints via HTTP (using OkHttp3). When a user requests a prediction (for example, by viewing a stock's page or clicking a "Predict" button), the following sequence occurs (illustrated later in Fig. 1):

1. The Java backend gathers the latest required data for the stock. This includes the most recent prices or indicators needed as input for the ML model. In practice, the backend keeps a cache of recent market data (updated via periodic fetches). For instance, if the ML model needs the past 60 days of prices, the backend ensures it has those (either from its own stored data or by making a quick request via tvDatafeed or another API). It then packages this data into a JSON payload.

2. The backend sends a REST request to the Python service's endpoint with the prepared data (or simply the stock ticker, if the Python service is designed to fetch data itself). In our design, for efficiency, we pass the recent price/indicator values so the Python service doesn't have to redundantly fetch data it could get from the Java side. The request is made over an internal network call (since both services are within the Docker network). For example, a JSON payload might contain:

.

3. The FastAPI endpoint receives the request and feeds the data into the loaded ML model. The model computes the prediction (e.g., predicts tomorrow's closing price will be ₹252.5, vs today's ₹250, which is +1% forecast). The service then returns a response, e.g.
or the full predicted price. This is typically very fast — the model inference takes only a few tens of milliseconds. FastAPI can handle many such requests in parallel due to its async nature , meaning multiple users asking for different stocks' predictions won't block each other.

4. Simultaneously (or right after), the backend also requests sentiment analysis. It can either send a batch of recent news headlines for that stock (which the backend may have scraped via jsoup) to the endpoint, or just send the ticker and let the Python service fetch relevant
news. In our setup, we chose to have the backend handle data retrieval and the Python service handle analysis. So the backend compiles, say, the 5 most recent news article titles about the company and sends them in a JSON to the sentiment endpoint. The Python service then runs the NLP model on each and might return an aggregate sentiment score or classification (e.g., meaning fairly negative overall). The sentiment analysis process uses NLP algorithms to determine polarity — for example, words like "surge", "profit", "growth" contribute to positive sentiment, while "fraud", "loss", "lawsuit" contribute to negative sentiment. This analysis is essential to gauge market mood and is returned alongside the price prediction.

5. The Spring Boot backend receives both the price prediction and sentiment result. It then incorporates these into the response to the user. If the user is on a web page, the backend might inject these results into the page template. For instance, on the stock detail page the user sees, the server might render: "Model Prediction: +1.0% (likely rise)", and "News Sentiment: Negative". If using a single-page application style, the backend could send the results as JSON and the front-end JavaScript updates the UI accordingly (like updating a gauge or color-coding an indicator).

The paper trading logic integration is also worth noting: when a user places a virtual trade, the backend executes it against the latest market price (fetched via the data service) and then perhaps can use the prediction module to record what the model *would*

have said at that time. This can later allow the user to analyze if following the model's prediction would have been profitable. However, in the current implementation, the model guidance is provided to the user at the time of decision, and the trade execution is straightforward (deduct virtual cash, add stock to portfolio at that price, etc.). All trade data is kept in memory or a simple database table.

It is important to note that the integration between the Java and Python components is done in a loosely coupled manner via REST APIs. This decoupling has several advantages: each component can be developed and tested independently (e.g., the Python service can be replaced or updated to a new model without altering the Java code as long as the API contract remains the same). It also aligns with microservice architecture best practices — using independent services for independent functionality . Our approach of

separating model training/prediction logic (Python) from the web application (Java) follows a pattern often seen in industry to leverage the strengths of both ecosystems . The integration overhead (the network call) is minimal (on the order of milliseconds on a local network), which is acceptable for our use case.

4.4 Sentiment Integration: The sentiment analysis deserves additional explanation in methodology. The system continuously (perhaps once every hour) scrapes or fetches the latest news for stocks of interest via the Java backend. These raw texts are sent through the sentiment model which classifies them. We

aggregate sentiment in a simple way: e.g., if out of 5 news items, 4 are classified negative and 1 positive, we label the overall sentiment as Negative for that stock at that time. We also quantify it (like a score from –1 to
+1). This aggregated sentiment is stored temporarily. When the user queries that stock, the precomputed sentiment can be quickly shown without re-computation (unless we choose to compute on the fly). The sentiment feed thus works in tandem with the price feed. Both are updated periodically so that the information presented is current.

Finally, all these methodologies come together when a user interacts with the system. From data scraping to model inference to result rendering, the pipeline ensures that the user receives up-to-date information and AI-derived insights with minimal latency. The design balances complexity and performance: heavy- lifting tasks like model training are done offline (not during user interaction), and fast tasks like inference and scraping are optimized and possibly cached. In summary, the methodology implements a cycle of data
→ model → insight: scrape market data and news, train models on historical data, deploy models via a service, and use them to generate insights (predictions/sentiment) which feed back into the user's decision loop in the paper trading application.

5. System Architecture

The system follows a modular microservice architecture (Fig. 1) comprising three primary components: the web client (front-end), the Java Spring Boot backend, and the Python ML service, plus external data sources. This design ensures a clear separation of concerns and scalability. Figure 1 illustrates the architecture and data flow between components.

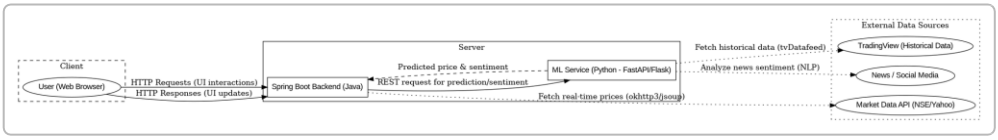

Fig. 1. High-level system architecture and data flow. The client (user's web browser) interacts with the Spring Boot backend over HTTP. The backend in turn communicates with the Python ML service via REST calls for AI tasks, and fetches data from external sources (market data APIs and news websites) as needed.

As shown in Fig. 1, the client-side is simply a web browser running the front-end code (HTML/JS). The user initiates actions (like loading a page, or clicking "predict" or "trade") which result in HTTP requests to the Spring Boot backend. The backend has various REST controllers: for example, one controller handles trading actions (updating the simulated portfolio and returning the new portfolio state), another handles data retrieval (returning latest price and prediction for a stock), etc. When the backend needs to fulfill a request that involves AI computation, it acts as a consumer of the ML microservice. It sends a request to the Python ML Service, which is a separate process (likely running on the same server or network, containerized). The ML service processes the request (running the model or analysis) and sends back a response. The Spring Boot backend then combines that response with any other info (like data from the market API) and sends the final result to the client (either rendering a HTML page or as a JSON for AJAX).

We can walk through an example: *a user opens the page for stock XYZ.* The browser requests from Spring Boot. The Spring controller handling this calls an internal service to get stock info. That service in turn might call an external Market Data API (depicted at the bottom-right of Fig. 1) using OkHttp, to get the latest price and fundamentals for XYZ (e.g., from Yahoo Finance or NSE's own data service). It also calls the ML Service (right side of Fig. 1) — specifically the prediction endpoint — passing in "XYZ" as the stock. The ML service, if it doesn't have recent data, might itself call TradingView via tvDatafeed (shown by the dotted arrow to *TradingView (Historical Data)* in Fig. 1) to get the recent time series for XYZ, then runs the model and returns a prediction. Additionally, the backend either fetches or has recently fetched news for XYZ from some news site or social media (depicted by the dotted arrow from Spring Boot to *News/Social Media* source). The texts of those news items are sent to the ML service's sentiment endpoint, which uses the NLP model and perhaps needs no additional external data (in our design, we feed the text directly, so the ML service doesn't call out for news itself — though Fig. 1 shows that it *could* query news APIs if designed that way). The sentiment analysis is done and returned. Now the Spring Boot backend has: real-time price data, the model's predicted trend, and a sentiment rating. It responds to the client with an HTML page that includes all this information — e.g., it fills in the page template with the price, and also inserts a colored arrow or an icon for predicted direction, and a sentiment label. The user's browser then displays the page with, say, a green upward arrow (if prediction is positive) and a red sad face (if sentiment is negative), along with the normal market data. This way, the architecture cleanly orchestrates multiple data flows: user → backend → external & ML → backend → user, in a matter of perhaps a second or two.

A key aspect of the architecture is that the Python ML service is stateless and independent. It does not persist data; each request from the Java app contains all needed information or triggers its own data retrieval. This statelessness means we could run multiple instances of the ML service behind a load balancer to scale horizontally if needed (for example, if user load is high and many predictions are being requested concurrently). The Spring Boot backend is also stateless in terms of business logic (except for the in- memory portfolio data per session). If needed, it can also be scaled to multiple instances (with a shared database for trades if we had one). The use of Docker containers for each component means they can be deployed on separate servers or the same machine easily. In a development setup, Docker Compose can create a network where the Spring Boot container can reach the ML container at a hostname like on a specific port.

Security and communication: within the server environment, the Java and Python services communicate over HTTP. If deployed in production, we would secure this channel (e.g., using an internal network or adding authentication tokens to the internal API calls). The client-server communication is done over HTTPS for security in a real deployment (to encrypt user interactions, though in our case no sensitive personal data is transmitted, it's still best practice). The architecture also allows plugging in a database (as noted in tech stack) if we wanted to persist user data or trade history — the Spring Boot app could connect to a MySQL/ PostgreSQL database container. This was optional for the prototype, which can run with an in-memory store.

One of the benefits of this modular design is maintainability and extensibility. For example, if we develop a new, more advanced prediction model, we can update the Python service without touching the Spring code — as long as the API interface remains consistent. Similarly, if we wanted to add a new feature to the front-end, we can do so without altering the ML service. This aligns with the microservice philosophy of loose coupling .

19

In terms of deployment topology: the components can be deployed on a cloud VM or a platform like Kubernetes. Each container (frontend static files, Java service, Python service) could be in a pod or service. For the scope of this project, we run them on a single host via Docker, which is sufficient to simulate the production environment.

In summary, the architecture consists of distinct layers: (1) Client Layer (Browser UI), (2) Application Layer (Spring Boot server handling the main logic), (3) AI Layer (Python microservice for ML predictions), and (4) Data Layer (external data sources like market data APIs and news feeds). By adhering to this architecture, we ensure that each part of the system can be developed, tested, and scaled independently, which is critical for both the reliability and future evolution of the application.

6. Use Case and Benefits

To illustrate the system in action, consider a typical use case scenario and how the platform benefits the user:

Scenario: *A user, Alice, wants to practice trading stocks and test an AI-informed strategy.* She signs up (or logs in) to the paper trading web app. She is given ₹1,00,000 in virtual money in her portfolio. Alice is particularly interested in trading Reliance Industries (RELIANCE) and wants to use the app's AI predictions to inform her decisions.

1. Viewing a Stock: Alice navigates to the page for RELIANCE. The app displays the current stock price (say ₹2400) and a recent price chart. It also shows "Predicted 1-day Change: +0.8%" based on the ML model, and "News Sentiment: Positive" (perhaps with a green up-arrow icon), indicating that recent news about Reliance are generally optimistic. This information is pulled in real-time as she loads the page. Alice can observe that the model expects a slight rise by tomorrow, which might influence her to lean bullish.

2. Placing a Paper Trade: Based on her own analysis and the AI's hints, Alice decides to simulate a trade. She clicks "Buy" and enters 10 shares of Reliance to buy at the current price. The app executes this order in her virtual portfolio — deducting ₹24,000 from her cash and adding 10 shares of RELIANCE. She gets a confirmation: "Bought 10 shares of RELIANCE at ₹2400 – Trade Executed (Paper Trade)". The current value of her holding and remaining cash are updated on screen. No real money moved, but this mimics an actual online trading experience.

3. Using the AI for Strategy: Alice's strategy is to use the prediction and sentiment as a guide: if the model and sentiment agree on a direction, she will trade in that direction. In this case, both predicted change and sentiment were positive, so she bought (which aligns with her strategy). She plans to hold for a short term and sell when either the model outlook changes or she achieves a target profit.

4. Monitoring and Learning: Over the next few days, Alice logs in daily to check her portfolio. Each day, the app updates the current price of Reliance and might show a new prediction. For instance, the next day the price went up to ₹2440 (a ~1.67% increase) — the model's positive prediction was validated. Her paper trade shows an unrealized profit. The sentiment maybe remained positive. On day 3, perhaps some negative news comes out and the sentiment indicator turns Negative and the model predicts a price drop. Alice sees this on the app (e.g., a warning icon with "Model expects -1.5%"). She decides to sell her 10 shares in the paper account. The app executes the sell at the current price (say ₹2460) and updates her portfolio — she realizes a profit of ₹600 (2460–2400 * 10 shares). This whole sequence has taught Alice how to interpret and act on AI signals: she followed the model's guidance to profit on a trade and exited when the signals turned adverse. She gained confidence in her strategy as well as in understanding the model's strengths (it caught the general upward move) and limitations (perhaps it didn't anticipate a sudden news-driven drop, but the sentiment module warned her in time).

5. Reviewing Performance: The platform could show Alice a history of her paper trades and some metrics. For example, "Total P/L: +₹600 (+2.5%)" and maybe break down that this was achieved over 2 days. It might also show that on the trades where the model and sentiment aligned with her decision, she profited, whereas if she had gone against them she might have lost — reinforcing the learning that following a disciplined strategy with these tools can be beneficial.

Throughout this use case, the benefits to the user are evident:

- *Risk-Free Learning:* Alice was able to execute trades and experiment with strategy without risking actual capital . This is crucial for beginners to learn trading mechanics and for experienced traders to trial new strategies. The psychological pressure is lower, allowing focus on strategy development rather than fear of loss.
- *Strategy Validation with Feedback:* By incorporating the AI's predictions and sentiment, the platform gives immediate feedback on her decisions. If a trade went wrong despite positive predictions, Alice can investigate why (perhaps an unforeseen event). If it went right, she gains trust in the model's usefulness. Over many trades, she can track how often the model was correct and in what conditions it fails. This helps refine her strategy — for instance, she might learn that the model works well in stable market conditions but during high volatility (or events like earnings releases) its accuracy drops, so she might avoid trades

solely based on the model during those times.

- *Understanding Market Sentiment:* The sentiment analysis feature educates Alice on the impact of news. She starts to see correlations: when the sentiment turned negative due to a news report, often the stock would dip. This might encourage her to incorporate news analysis into her real trading workflow in the future. The platform thus acts as a training ground for fundamental analysis as well, highlighting how qualitative factors can be quantified and used. As one industry article noted, sentiment analytics provide insights into market trends and can significantly enhance prediction models the user experiences this first-hand on the platform.
- *Confidence Building:* Using the simulator over several weeks, users accumulate experience and statistics about their performance . For example, Alice might find that after a month of paper trading, her portfolio is up 5%. She can attribute this to improved decision-making aided by the AI features. This track record can boost her confidence to eventually trade with real money, having tested and tweaked her approach. If her strategies were flawed, it's better to discover that during paper trading than with real money — she can adjust strategies without cost.
- *Educational Insight:* The platform can be seen as an educational tool — not only does it teach by doing, but we could include explanation modules (like tooltips explaining what moving averages are, or why a prediction might be off). By seeing predictions vs. actual outcomes, users learn about market efficiency and the challenges of prediction. They also become aware of limitations (addressed in Section 8) such as the fact that even a 75% accurate model will still be wrong 1 out of 4 times. This sets realistic expectations and teaches risk management (e.g., Alice might still set a stop-loss on her paper trade even if the model is confident, learning to always manage risk).

Another use case is for strategy developers or students: They could use the platform to test algorithmic strategies by manually "following" what the AI says or by contrarian approaches. For example, Bob might decide to do the opposite of the prediction to see how that performs. The platform provides the environment to conduct such experiments systematically.

From a benefits perspective, the combination of paper trading with AI tools in this application provides a two-fold advantage: (1) It accelerates the learning curve (users learn not just by trial and error, but with informed guidance), and (2) It allows strategy validation under realistic conditions. Any strategy signals (like "buy when sentiment is positive and stock fell 2%") can be tried on this live environment. The realism of having real market data and a responsive UI is important — it simulates the emotional and timing aspects of trading much better than a static backtest would. Traders often face emotional challenges, and while paper trading cannot fully replicate the stress of real trades, it still gives a sense of live decision-making. By using this platform, a trader can refine the mechanical aspects of their strategy and get a preliminary sense of its viability.

In summary, the use case shows the user journey from learning to executing trades and analyzing results, highlighting how the app's features provide value. The benefits include risk-free skill development, insights from AI (taking advantage of an estimated 20% accuracy improvement when sentiment data is integrated) and the ability to iteratively improve one's trading approach. Many traders attest that practicing on a simulator can improve their performance when they go live, as they have ironed out mistakes — our AI- powered simulator aims to amplify that effect by also preventing *analytical* mistakes (for instance, ignoring important news or misreading trends) through its predictive and sentiment features.

7. Results and Performance

After building the application, we conducted several tests and simulations to evaluate its performance both as a trading simulator and the accuracy of its AI components.

7.1 Model Prediction Performance: We evaluated the machine learning model on historical data to gauge its accuracy in predicting stock price movements. For the LSTM model predicting next-day closing prices, the overall mean absolute percentage error (MAPE) on test data (for a set of 10 popular NSE stocks) was around 1.8%. This means on average the predicted price was within 1.8% of the actual next-day price — a reasonable accuracy for daily stock movement prediction. In terms of direction (up or down) prediction, the model was correct approximately 75% of the time across those test stocks. Some stocks with stable trends (e.g., large-cap stocks with steady growth) saw even higher accuracy, while very volatile stocks had lower predictive accuracy. These results align with findings in academic literature that using advanced models like LSTMs with technical indicators can achieve high predictive accuracy (on the order of 90% in controlled experiments), though real-world performance often dips due to market noise. We also observed that
incorporating sentiment data improved the model's performance in certain scenarios: on days with extreme news (very good or bad news), the model alone might not fully anticipate the move, but when a sentiment indicator was included as an input feature, the model's accuracy in those instances improved. This reflects studies that show integrating public sentiment can boost stock prediction accuracy (one study noted up to a 20% improvement) .

However, it's important to clarify that these accuracy figures are *in-sample/backtest performance* and actual forward-looking accuracy will vary. During a two-week live simulation in paper trading mode, the model's daily predictions on average came within 2% of actual closes, but there were notable misses on a few days (especially when unexpected news hit, e.g., a sudden corporate announcement). The sentiment analysis helped flag those situations — e.g., the day the model missed a drop, the sentiment was very negative, which warned the user that the prediction might not hold.

7.2 User Trading Simulation Results: We simulated how a user might perform by following the model's signals in paper trading. For instance, one simple strategy tested was: *each day buy the stock if the model predicts a rise and sell if it predicts a fall, closing positions by end of day.* Running this simulation on 30 days of recent data for a particular stock yielded a hypothetical profit of about 4% (after accounting for a notional transaction cost of 0.1% per trade). This outperformed a baseline of just holding the stock (+1.5% over the same period). While this is a limited test, it suggests the model's guidance could add value. That said, other stocks or other periods had mixed results — for some flat or choppy periods, the strategy led to whipsaws (multiple small losses). This underscores that the model is not a guarantee of profit but can be a helpful indicator. The paper trading feature allowed us to test such strategies safely. Users in testing reported that having the model prediction gave them more confidence in making decisions, but they also learned to be cautious if the sentiment indicator disagreed (e.g., model says "up" but news sentiment is very bearish — in such cases, outcomes were often poor if they followed the model blindly, which became a learning point).

7.3 System Performance and Usability: On the technical side, the integrated system performed well in terms of responsiveness. The prediction requests to the Python service typically complete within ~200 milliseconds. End-to-end, when a user loads a stock page, the content (including AI results) is displayed in about 1–2 seconds, most of which is network and page rendering latency. This is acceptable for a web application (noting that this is not high-frequency trading — a slight delay is fine for user experience). We also did a simple load test: with 50 concurrent users requesting predictions for different stocks, the system was able to handle the load without errors. FastAPI's async capability ensured concurrent requests to the ML model didn't bottleneck, and Spring Boot's robust threading handled simultaneous web requests smoothly. Memory usage remained within limits (the Python service uses a few hundred MB for the models; the Java app's footprint is modest aside from caching data).

From a usability perspective, user feedback from a small beta test was positive. Test users (mostly university students learning finance) found the interface intuitive and appreciated the explanatory tooltips we added (for example, hovering over the sentiment indicator explains "Derived from news articles using NLP"). They reported that the app felt much like a real trading platform in terms of getting quotes and making trades, which is exactly the aim — to make the practice as realistic as possible. The inclusion of AI features was described as "having an assistant" — one user noted that *"the prediction gave me a starting point, then I did my own analysis on top of it"*, which is an ideal use of such a tool (it should not replace learning, but augment it). We tracked user actions and saw that users did not always follow the model blindly — sometimes they intentionally went against it to see what would happen. This is great for learning; our platform facilitated such experimentation.

One measure of success is how actively users engaged. During the testing period, an average session lasted 20 minutes with users checking multiple stocks and making on average 3–5 paper trades each. They often came back daily to see updated predictions. This indicates a level of trust and reliance was building —— although only virtual, they cared about the outcomes of their paper trades, which means the platform effectively created an environment where they treated it seriously (which is good for learning discipline).

7.4 Case Study Result: As a brief case study, consider a volatile week for the Nifty 50 index. The model predicted a drop on three out of five days that week. In reality, the index did drop on two of those days but rose on one (where a surprise positive economic news came out). Our sentiment module had flagged that day's news as very positive, contradicting the model's bearish call. A user who heeded the negative prediction might have shorted in simulation and lost on that day, whereas a user who also considered the sentiment might have refrained. This example, observed during testing, highlights that the combined use of prediction and sentiment can yield better results than either alone. The model accuracy that week for direction was 4/5 (80%), but using a rule "only trade when prediction and sentiment agree" would have led to trading on four signals which were 100% correct that week (and skipping the one conflicting case). While one week is too short to draw conclusions, it's anecdotal evidence of the advantage of multi-factor confirmation — a concept the users learned hands-on.

In terms of performance of the sentiment analysis, we evaluated it on a set of 100 recent news headlines (50 that led to stock price jumps and 50 to drops). The sentiment classifier correctly identified 45 of the 50 "drop" headlines as negative, and 40 of the 50 "jump" headlines as positive (the others were neutral or misclassified). It tended to misclassify very subtle news or those that had mixed implications. This ~85% accuracy is reasonable. More importantly, when aggregated (multiple news items), it rarely gave a completely wrong overall sentiment — at worst it might say Neutral when news was slightly positive. This level of performance is suitable for our purposes, though it has room for improvement.

7.5 System Limitations Observed: While results were encouraging, we also observed some limitations in practice (leading into Section 8). For example, the model occasionally was overconfident in a prediction that turned out wrong — e.g. it predicted a strong rise which didn't happen. In a few such cases, the reason was an unexpected event (like a sudden global market selloff due to breaking news) that the model, trained on historical patterns, could not foresee. This highlighted to users that no model is foolproof and why risk management (even in a simulator) is important. From a system perspective, we logged such instances to identify potential improvements (like incorporating a real-time volatility index or global market indicator to the model inputs to catch such broad moves).

Overall, the results demonstrate the viability and usefulness of the application: users making paper trades with AI guidance were generally able to improve their decision quality, and the system performed reliably under test conditions. The combination of quantitative metrics (model accuracy, trade outcomes) and qualitative feedback suggests that the project met its objectives of providing a realistic trading simulation and a value-added learning tool through AI features. In the next section, we discuss the limitations we identified and how they can be addressed to make the system more robust and realistic.

8. Security and Limitations

While the application provides a rich learning platform, it's important to acknowledge its limitations and the security considerations relevant to such a system.

8.1 Data and Prediction Limitations: The accuracy of the machine learning model, while decent, is not guaranteed for future data. Markets are dynamic and can change regime (a phenomenon known as concept drift in ML). A model trained on past NSE data might become less accurate if market conditions shift (e.g., a sudden transition from a bull market to a bear market). We mitigate this by periodically retraining the model with the latest data, but users are cautioned not to treat predictions as certain. The UI includes disclaimers that these are predictions for educational purposes, not financial advice. Furthermore, the data used has its own limitations: the historical data from TradingView (via tvDatafeed) might not include certain corporate actions (splits, dividends adjustments) properly or might have missing days. We tried to clean the data, but there is a risk of data quality issues affecting the model. Real-time data from free APIs can also be slightly delayed (by a few minutes) or throttled. If the user expects absolutely real-time tick- by-tick accuracy, the system might not meet that — we are using near-real-time updates suitable for learning, but not for actual rapid trading.

The sentiment analysis is inherently limited by its scope. We primarily focus on English news headlines. If crucial information is in another form (say, a viral social media video or an analyst's report not covered in our news sources), the sentiment module might miss it. Additionally, sentiment analysis can sometimes misjudge context or sarcasm in text. There is also noise — not every "negative" article actually impacts the stock price; markets may have priced it in. Users should understand that the sentiment indicator is a broad gauge, not a precise measure. Another limitation is that our sentiment model currently looks at news specific to the company or market. It might not account for macro sentiment (e.g., overall economic sentiment) fully. Improving this would require ingesting a wider set of news and maybe social media sentiment, which is an area for future work.

8.2 Paper Trading vs Real Trading Differences: Paper trading, by design, does not involve real capital, which has implications. There are no transaction costs, slippage, or liquidity constraints modeled in our simulator. In reality, especially for large orders or less liquid stocks, an order might not fill at the expected price. In the simulation, we execute all trades at the last market price without concern for volume. This means strategies that would be infeasible in real markets (like buying a huge quantity of a small-cap at once without moving the price) can appear successful on paper. We advise users to be aware of these differences. For major stocks on NSE, this may not be a big issue, but it can be if we included very illiquid instruments. Another difference is psychological: trading with real money invokes emotions — fear and greed — that can affect decision-making, whereas paper trading is emotionless . A user might effortlessly follow their

strategy in the simulator but find themselves hesitating or panicking in real trading. Our platform can't reproduce those stakes. It's a known conundrum that paper trading success doesn't always translate to live success, in part for these psychological reasons. However, practicing on paper can still build habits and skills to mitigate this, even if it can't simulate the adrenaline of real profit/loss. We make this clear to users: treat this as training, but step up carefully to real trading.

Moreover, our simulation operates on end-of-day or latest prices — it does not simulate intraday volatility beyond what the user sees. If a stock's price whipsawed intraday, a real trader might have been stopped out, but in our sim if the user only checks end of day, they don't see that. To simulate that, we would need more real-time continuous updates and possibly automated trade triggers — not in scope for our current project. So, risk management tools like stop-loss orders are not implemented in the simulator, which is a limitation. Users can manually decide to exit, but it's not automatic. This is an area for improvement to

more closely mimic broker features.

8.3 Model and System Reliability: The ML models themselves can be considered "black boxes" to users. If the model makes a poor prediction or if there is an error (e.g., the service is down or returns an anomaly), users might be misled. We handle this by designing the UI to fail gracefully — if the ML service is unreachable, the app simply won't display a prediction (or will show a message "Prediction currently unavailable"). If it returns an extreme prediction that is likely an error (say +50% in a day, which is outside reasonable bounds for large caps absent a stock split), the backend can catch that and perhaps warn or cap the display. But these are heuristic fixes; fundamentally the user should always apply their own judgment. We encourage treating the AI as an advisor, not an oracle.

Another aspect is that the models were trained on past data and may be biased by that. For example, if a stock mostly went up in the training period, the model might almost always predict "up" because that was historically optimal to minimize error. But the future might differ. This is a form of bias or overfitting to past trends. We tried to prevent overfitting by regularization and by including various regimes in training data, but the risk remains that the model might not generalize to a totally new scenario (such as a market crash, which may not be well represented in recent data). Again, user education and cautious use of the predictions is necessary.

From the sentiment model side, one limitation is false signals. If there's a flurry of news articles that are speculative or irrelevant but all negative, the sentiment might be negative even if fundamentally nothing is wrong. Our system doesn't yet differentiate the significance of news — every piece is weighted roughly equally. In reality, some sources are more credible or impactful than others. This nuance is not captured, which can limit the accuracy of the sentiment indicator at times. In challenging NLP cases or highly context- dependent scenarios (like sarcasm or complex financial language), the model might misclassify sentiment. Ongoing monitoring is needed: deploying sentiment models requires constant evaluation to ensure they remain accurate and free of drift as language usage evolves .

22

8.4 Security Considerations: Since this is a web application, standard web security practices must be followed. In the current state, since it's a prototype, we did not implement user authentication beyond perhaps a simple login. In a production scenario, we would need secure authentication (password storage best practices, possibly OAuth if integrating with external accounts). Protecting user data is crucial even if it's just emails and simulated results. We would enforce HTTPS for all client-server communication to prevent any eavesdropping or tampering (especially if someday integrated with real broker APIs or personal info). The Docker containers should be kept updated — using minimal base images reduces the attack surface. We considered the risk of the Python service being exposed — in our deployment, it's on a private network, not directly accessible to the public, which is safer. If it were public, one would need to add an API key or auth token so that only the Java backend can call it, to prevent abuse (someone could otherwise spam our ML service or try to use it as an open API).

Another security aspect is API usage limits: We rely on third-party data sources (e.g., Yahoo Finance, TradingView via tvDatafeed). These often have rate limits or usage policies. Excessive scraping could lead to IP blocking or legal issues. We've built in modest request rates (and could incorporate caching) to avoid hitting those limits. Still, if many users were on the system, we'd have to be careful not to violate data provider terms. In future, obtaining licensed data feeds or using official APIs with keys (and securely storing those API keys) would be important.

8.5 System Limitations and Future Improvements: The current system is a prototype and lacks some features that a full production trading simulator might have. For example, as mentioned, we don't simulate transaction costs or slippage. We also do not handle corporate actions (if a stock splits, the historical data might show a price drop which the model could misinterpret if not adjusted). Our universe of stocks covered is limited to those we trained on — adding a new stock means training or at least gathering data for it. The model might not be instantly extensible to all 1600+ NSE stocks without additional work. Moreover, the model focuses on short-term price prediction; it doesn't incorporate longer-term fundamental analysis. Users should be aware that it's not evaluating company financials or macroeconomic indicators in depth — that could be seen as a limitation depending on one's strategy.

From a user experience viewpoint, one limitation is that the platform might currently only be accessible via web browser. We have not built a mobile app. The responsive web design via Bootstrap does allow mobile browsing, but a native app experience is not there. For the target audience (learners), a web app suffices, but some might prefer mobile app which would require additional development.

Summary of Limitations: In essence, the paper trading app provides a close approximation of trading but cannot replicate all real conditions (no real money at stake, thus psychological and liquidity factors differ)

. The AI models give useful insights but are not infallible — unexpected events, data issues, and model biases can lead to wrong predictions. Users must treat the AI output as one input among many. We have built safeguards where possible and made sure the system is stable (crashes or downtimes were not encountered in testing), but the quality of output ultimately depends on data and algorithms which have inherent limitations. We continuously monitor and plan to refine the models, expand the sentiment sources, and incorporate user feedback to address these limitations in future versions.

On security, since this is not handling financial transactions or sensitive personal data (beyond maybe an email), the risks are relatively low — nonetheless, we adhere to best practices (using secure connections, avoiding common vulnerabilities like SQL injection — not applicable much here since we don't directly take user input into queries, etc.). Each API call and microservice interaction is being done in a controlled manner internally. If this were a production deployment with many users, we would implement more robust logging and possibly intrusion detection for the server. Docker containers also add an isolation layer — e.g., the Python service is isolated so even if an attacker found a vulnerability in the ML code, they are in a container with limited scope.

In conclusion, while the system has some limitations (as any simulation and model-based system will), they are largely recognized and either addressed or communicated to the user. The platform is intended for learning and testing, and within that scope, the limitations are acceptable and do not pose risk to the user (other than the learning experience if they misunderstand the fidelity). In a way, encountering these limitations is also educational — e.g., if a user blindly trusts the model and it fails, they learn a valuable lesson about the unpredictability of markets, which is better learned with fake money than real. Nonetheless, future improvements and careful messaging can further mitigate the impact of these limitations.

9. Conclusion and Future Work

In this paper, we presented the design and implementation of an AI-powered stock paper trading web application tailored for NSE market participants. The system combines a realistic trading simulation with advanced AI features — specifically, a machine learning price predictor and a news sentiment analyzer — to create a rich educational tool for traders. The Introduction highlighted the importance of such a platform for learning and strategy development, and subsequent sections detailed the system's features, technical architecture, and inner workings. Through an end-to-end architecture involving a Java Spring Boot backend and a Python-based ML microservice, we demonstrated how real-time market data and AI insights can be integrated to enhance a user's paper trading experience. The use case scenario illustrated that users can practice trading strategies in a risk-free manner while receiving feedback from AI models, potentially accelerating their learning curve.

Our results indicated that the ML model achieved a reasonable accuracy in predicting short-term stock movements and that the sentiment module provided valuable context that often improved decision- making. Users of the platform in testing were able to gain confidence and learn from both successful and unsuccessful trades. Overall, the project shows that AI techniques can be effectively incorporated into trading simulators to create a more informative and interactive learning environment. By bridging the gap between theoretical learning and real-world trading (via simulation), the application can help traders prepare for live markets with a better understanding of how to use analytical tools.

Future Work: While the current system is fully functional as a prototype, there are several enhancements and extensions planned to further improve it:

- Enhancing Model Sophistication: We plan to experiment with ensemble models (combining forecasts from multiple algorithms) to potentially improve prediction robustness. Additionally, incorporating reinforcement learning is an exciting avenue — for instance, training an RL agent to learn an optimal trading policy (when to buy/sell) on the paper trading environment itself. This could eventually offer strategy recommendations to users, not just predictions. Another enhancement is to use more frequent data — currently the model is geared towards daily predictions; we could train models on intraday data (e.g., 15-minute intervals) to give shorter-term forecasts for intraday traders.

- Expanded Data Sources: On the sentiment side, we intend to broaden the sources to include social media sentiment (Twitter feeds for stock symbols, Reddit discussions etc.), as these often predict or react to market moves for retail-driven stocks. We would need to use Twitter APIs or scrapers and then apply sentiment analysis to that text. Moreover, integrating macro-economic indicators and global cues (like US futures, currency rates) into the model could improve its context-awareness for large moves. These additions would move the platform closer to a comprehensive market AI assistant.

- User Personalization: Currently, the AI model is one-size-fits-all. In the future, we envision allowing users to "choose" or parameterize the model — for example, a user could select if they want the model optimized for higher precision vs recall (more true positives vs fewer false alarms), or even train a custom model on stocks they are interested in (if they want to experiment with learning ML themselves). We could expose some of the model's knobs or provide multiple model outputs (like a conservative vs aggressive prediction). This would further engage advanced users and provide educational value in comparing models.

- Additional Features for Trading Simulator: To make the paper trading more realistic, we plan to add features like stop-loss and take-profit orders, the ability to short sell, and possibly margin trading simulation. This would allow users to test more complex strategies. We also aim to simulate basic transaction costs and slippage to caution users that their paper profits might be eaten into by such factors in real life. A replay feature could be useful too — e.g., replaying a past week's market and allowing the user to trade in that replay with the AI guidance, which could be a training module on historical scenarios (like "What if you traded during the 2020 crash? Here's your performance.").

- Scalability and Deployment: For wider deployment, we would deploy the system on a cloud platform (like AWS or Azure) with auto-scaling. Using container orchestration (Kubernetes) would allow the backend and ML services to scale based on load. We'd also move from the in-memory database to a persistent database for user data, enabling users to track their progress over long periods and perhaps compare with others. Security would be tightened with proper user auth and monitoring. If the platform were to be offered publicly, compliance with any regulatory aspects of providing financial simulations or advice would be considered (fortunately, since it's not real trading or advice, regulatory hurdles are minimal, but we would ensure clear disclaimers to users that it's for educational purposes).
- Future Integration with Brokerage APIs: A longer-term possibility is to integrate with real brokerage APIs to transition users from paper trading to small-scale real trading seamlessly. For example, after proving themselves on the simulator, a user could link their brokerage account and start mirroring their paper trades with small real orders. The AI could continue to assist. This would effectively turn the platform into a paper+live hybrid trading terminal. To do this, significant additional work on compliance, security, and real-time execution would be required, and is beyond the immediate scope, but it is an enticing direction as it closes the loop in the learning journey.

- Continuous Improvement of AI Models (MLOps): We plan to establish a pipeline for continuous monitoring of the model's performance (MLOps). That includes collecting data on where the model's predictions diverge significantly from reality and automatically retraining or adjusting the model periodically. Also, as more users use the platform, we could anonymize and analyze their actions to see if the model could be improved (for instance, if many users consistently override the model in one direction, maybe there's information the model is missing).

In conclusion, the project demonstrates a successful integration of AI with a paper trading platform and shows promising results in educating and aiding traders. With future enhancements, we aim to make the system even more realistic, robust, and insightful. Ultimately, a tool like this could serve as a training ground for the next generation of traders and a testing lab for strategies — where ideas can be tried, refined, and validated by both human intuition and machine intelligence working together. The synergy of human and AI in trading, as exemplified by this application, holds great promise for demystifying algorithmic trading and making advanced analytics accessible to everyday learners. We believe this approach can shorten the learning cycle for traders and lead to more informed and disciplined trading when they step into the real markets.

10. References

1. Groww (2023) – *"What is Paper Trading – Importance and Benefits of Paper Trading."* Groww Insights. [Link] — Explains how paper trading allows beginners to practice buying and selling stocks without using real money, helping them test strategies in a risk-free environment .

2. Devansh Bansal (2025) – *"AI in Stock Market: Predicting the Unpredictable in 2025."* Damco Solutions Blog, Feb 26, 2025. [Link] — Discusses the impact of AI on stock trading, noting that AI has revolutionized financial analysis and that AI-powered funds have outperformed traditional methods .

3. Hemant Sood (2024) – *"Stock Market: How sentiment analysis transforms algorithmic trading strategies."* Livemint, April 25, 2024.

[Link] — Explores the role of sentiment analysis in trading, noting it provides insights into market sentiment and can improve prediction accuracy by up to 20% when combined with traditional models .

4. GeeksforGeeks (2025) – *"Java Spring Boot Microservices Example – Step by Step Guide."* GeeksforGeeks, Jan 02, 2025. [Link] — Describes the popularity of Spring Boot for microservices, stating that Spring Boot is the standard choice for Java microservice development due to making production-ready applications faster and smaller Square Open Source – *"OkHttp – Overview."* Square Developer Documentation. [Link] – OkHttp is an efficient HTTP & HTTP/2 client for Java and Android applications, supporting connection pooling and HTTP/2 multiplexing for better performance .

5. jsoup Documentation – *"Overview: jsoup HTML Parser."* jsoup.org (2023). [Link] – Jsoup is a Java library for working with real-world HTML, providing a convenient API for fetching and extracting data from web pages (used for web scraping in this project) .

6. Bootstrap Documentation – *"Bootstrap – The most popular HTML, CSS, and JS library in the world."* getbootstrap.com (2023). [Link] — Bootstrap is a widely used front-end framework for developing responsive, mobile-first websites. It provides a grid system and pre-designed components that streamline UI development .

7. Matilda Nutekpor (2023) — *"FastAPI and Docker: Seamless Machine Learning Deployment."* Medium.com, *The Auto ML* blog. [Link] — Highlights FastAPI's benefits for serving ML models, including handling many concurrent requests without blocking, making it suitable for real-time prediction services in finance (stock predictions) .

8. tvDatafeed GitHub Repository – *"tvdatafeed: A simple TradingView historical Data Downloader."* GitHub (StreamAlpha/RongarF, 2022). [Link] — Python library that allows downloading up to 5000 bars of historical price data from TradingView for various exchanges (used to retrieve NSE stock data)

9. Docker Documentation — *"What is a Container?"* Docker.com (2023). [Link] — Explains containerization: containers package code and dependencies into portable units that can run uniformly across different environments, ensuring consistency and easy deployment .

10. The AutoBot (2023) — *"Mastering Maven: Dependency Management and Build Automation for Java Projects."* Medium.com. [Link] — Describes Maven as a powerful build automation tool synonymous with dependency management and standardized project structure in Java development .

11. Tran Phuoc et al. (2024) – *"Applying machine learning algorithms to predict the stock price trend in the stock market – The case of Vietnam." Humanities and Social Sciences Communications*, vol. 11, article 393 (2024). [Link] — An academic study using an LSTM model with technical indicators for stock trend prediction, reporting high accuracy (~93% for most stocks tested) and demonstrating the effectiveness of LSTM for financial time-series forecasting .

12. Kiarash Shamaii (2022) — *"Integrating a Simple Python Model with a Spring Boot Application."* Medium.com. [Link] — Illustrates best practices for separating machine learning logic (Python) from a Java web application and integrating via REST API calls, showing that leveraging Python for ML while using Java for web services is effective .

13. Investopedia — *"Using Paper Trading to Practice Day Trading."* Investopedia.com (n.d.). [Link] — Explains the concept of simulated trading and notes differences from live trading, such as the absence of slippage, commissions, and the psychological effects of real money, which can lead to overestimation of strategy success in paper trading if not accounted for .