Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Stock Saga: A Gemini Integrated Financial Trading Simulator

Kavya Vasa

kavya.vasa@somaiya.edu

Somaiya Vidyavihar University, K.J. Somaiya School of Engineering https://orcid.org/0009-0008-3279-5991

Dr. Mansi Kambli

Somaiya Vidyavihar University, K.J. Somaiya School of Engineering

Dr. Bhakti Palkar

Somaiya Vidyavihar University, K.J. Somaiya School of Engineering

Research Article

Keywords: Trading Simulator, Serverless Edge Computing, Algorithmic Backtesting, Pattern Recognition, JAMstack, Latency

Posted Date: April 14th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9380893/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Stock Saga: A Gemini Integrated Financial Trading Simulator

A Comprehensive Framework for Financial Education and Algorithmic Strategy Validation

Kavya Vasa[1*], Dr. Mansi Kambli (Project Mentor)[1] and Dr. Bhakti Palkar (Project Expert)[1]

[1]Department of Computer Engineering, K.J. Somaiya School of Engineering, Mumbai, India.

*Corresponding author(s). E-mail(s): kavya.vasa@somaiya.edu; Contributing authors: mansi.mk@somaiya.edu; bhaktiraul@somaiya.edu;

Abstract

Intraday traders are severely disadvantaged by speed. In our experience, one of the key reasons they slippage across trades is that their workflows are disjointed. A typical retail setup requires the trader to switch between a charting website, a separate broker terminal, and a social media news feed. Every click between tabs costs money. We propose an engineering solution, Stock Saga, which is a proprietary web platform that we built to solve the problem to bringing dis- parate trading tools together into a single high-velocity interface. In order to provide a real-time, financial data processing application without interface lock- ups, we designed the backend service architecture away from the conventional centralized monolithic servers and used a highly concurrent JAMstack archi- tecture on distributed Deno edge nodes. We also add a custom Convolutional Neural Network (CNN) integration in the charting DOM to detect trading pat- terns automatically on the client side, paired with Language Model inference pipeline to score breaking news. Before risking capital, users validate their algo- rithmic concept with our mathematical backtesting engine. We also present a custom Know Your Customer (KYC) pipeline that uses facial geometry vectors to verify identity documents without manual review. Our empirical stress tests show that this edge-routed architecture reduces API response latency by 71.9% compared to traditional centralized architectures and maintains browser mem- ory footprints under high-frequency WebSocket updates. In the end, this work presents a professional-grade system to backtest and run traders' strategies with zero UI latency.

Keywords: Trading Simulator, Serverless Edge Computing, Algorithmic Backtesting, Pattern Recognition, JAMstack, Latency.

1 Introduction

Intraday trading is all about timing. In a highly volatile environment, the opportunity window to ride a price move can be very narrow. A trader has to be glancing at live prices, tick by millisecond, as they interpret abrupt economic news flashes and execute a complicated order matrix within seconds.

1.1 The Reality of Execution Latency

The software ecosystem that is provided to retail traders on a daily basis is fundamen- tally broken because it is terribly disjointed. A normal home trader sits at a desk in front of three monitors. One displays a web-based charting software like TradingView. Another displays the actual brokerage terminal where they can place their order sizes. And a third one is usually an unfiltered social media feed or a news aggregator like Bloomberg.

We can quantify the time between recognizing an opportunity and actually entering the trade in the order book as execution latency, Δt_{exec}. In a multi-monitor, multi- software environment, this delay grows exponentially:

$$\Delta t_{exec} = \tau(T_c) + \tau(T_n) + \tau_{switch} + \tau(T_e) \tag{1}$$

Here, T_c is the cognitive time reading the technical chart and T_n is the time to read the news headline. The physical delay of moving the mouse from one screen to the other, clicking the correct browser window, and typing the number is τ_{switch}. And the time for the legacy broker's server to acknowledge the order is T_e.

During times of high execution latency, traders suffer from catastrophic market slippage (S_L). The slippage is the monetary gap between the price you wanted to buy at and the price you were handed when the slow order eventually got executed:

$$S_L = |P(t + \Delta t_{exec}) - P(t)| \tag{2}$$

At present, forcing users to bounce between a number of disconnected software tools, artificially expands τ_{switch}. S_L becomes much worse. It also adds in basic human errors. For example, when a trader is panicking to catch a 5% spike in a stock, it is easy to click the wrong button (”Sell” instead of ”Buy”) or to type an extra zero in their position size.

1.2 Original Contributions of this Research

We became aware that it was not sufficient to just try to make incremental improve- ments to existing tools. We needed to create a single system from the ground up that optimistically eliminates every possible source of Δt_{exec}. In particular, this paper introduces four novel engineering contributions:

1. We built a dedicated JAMstack infrastructure on top of Deno edge nodes which effectively removed physical data travel time and the T_e bottleneck for the end user.
2. We built a local, low-latency backtesting engine which forces retail traders to backtest their strategies with institutional risk math (ie. Max Drawdown) before deploying them.
3. We circumvented classic server-side AI bottlenecks by employing very small deep learning models directly on the client side. A CNN surveys the live canvas for visual patterns, and an LLM constantly consumes the news feed for sentiment.
4. We designed an automated identity verification pipeline using spatial vector map- ping to instantly approve new users, skipping the typical 3-5 business day waiting period that legacy brokers endure.

2 Related Work and Baseline Methodologies

We had to first understand in what ways the attempts at financial software engineering that came before us broken down. We didn't want to reinvent the wheel on the math for basic neural networks, but we quickly learned that the academic models don't survive deployment in real trading environments.

This was the first attempt by researchers like Bao et al. using Convolutional Neu- ral Networks (CNN) and Long Short-Term Memory (LSTM) nodes to forecast stock trends. The main contribution was demonstrating that a 2D CNN could view a stock chart as an image and detect the geometric shapes in a chart (e.g. flags or triangles) just like a human analyst, while completely ignoring the raw numerical price data.

Later, researchers like Akita et al. showed that adding Natural Language Process- ing (NLP) to read the financial news indeed improved the predictions because markets respond to human emotion, not pure math.

But the big problem with almost all of these prior works is the deployment archi- tecture. In academic settings, these gigantic CNN and LSTM models are deployed on giant desktop GPUs or dedicated AWS clusters. If a vanilla web developer tried to squeeze a standard 500 megabyte Python-based deep learning model on a React web app, the user's browser tab will lock up. The DOM can't be used to update a live stock chart at 60 frames per second while doing matrix multiplications for a neural network. Additionally, traditional financial applications depend on monolithic servers. Pre- vious work in financial sentiment analysis also used server-side batch processing. A news article goes in, the server spends five minutes thinking about it. Then it pushes a sentiment score to the user. In intraday trading, five minutes is too late. The price has moved and the opportunity is gone.

Our original contribution focuses specifically on these structural weaknesses. We posited that if we could offload the computational burden from a single centralized

server to a decentralized “Edge” network, we could give the benefit of these highly complex AI models to users without the need to freeze their trading charts or experience excruciating latency spikes.

3 Architectural Design and Edge Routing

If a trading app crashes, stutters, or loses the WebSocket connection for even a few seconds, a trader can lose a sacrificial amount of capital. For us, system stability in the face of heavy computation was the number one priority. In this section, we explain how we discarded the typical web server model entirely in order to deal with a high volume of data.

3.1 The Monolithic Server Bottleneck

Most legacy platforms use standard monolithic servers. A company will rent a massive AWS EC2 instance in some single geographical chokepoint (e.g. some data center in Ashburn, VA). If C is our client machine, S is the centralized server, and M is the external market data API that provides the stock prices, then the data request path is as follows: $C \rightarrow S \rightarrow M \rightarrow S \rightarrow C$.

The total latency delay experienced by the trader is calculated as:

$$L_{traditional} = 2 \cdot d(C, S) + 2 \cdot d(S, M) + t_{proc} \quad (3)$$

where $d(x, y)$ is the physical distance the fiber-optic data travels and t_{proc} is the time spent parsing the request by the server. If a retail trader in Mumbai wants a live stock quote from a central server in New York, the physical distance ensures horrible lag. We solve this with a Deno Serverless Edge architecture.

3.2 Deploying V8 Isolates at the Edge

Instead of having one server, our backend logic is compiled into very small, independent functions that exist on a global CDN network. Deno executes these functions via Google’s V8 Javascript engine isolates. An isolate is essentially a microscopic sandbox that boots in under 5 ms.

Now imagine a user in Mumbai opening a chart. The request does not go to New York. It is caught immediately by an edge compute node in Mumbai, E_i. The new routing path is $C \rightarrow E_i \rightarrow M \rightarrow E_i \rightarrow C$. With $d(C, E_i)$ almost zero, the wait is:

$$L_{edge} = 2 \cdot d(C, E_i) + 2 \cdot d(E_i, M) + t_{proc} \quad (4)$$

The primary reason our platform feels as fast as a native desktop application running on top of a web browser is that we ensure that the L_{edge} is only a fraction of $L_{traditional}$

3.3 State Management and Persistent Sockets

Serverless edge functions are awesome for latency, but they come with one catastrophic engineering back pain: the cold start problem. If a particular trading function hasn’t

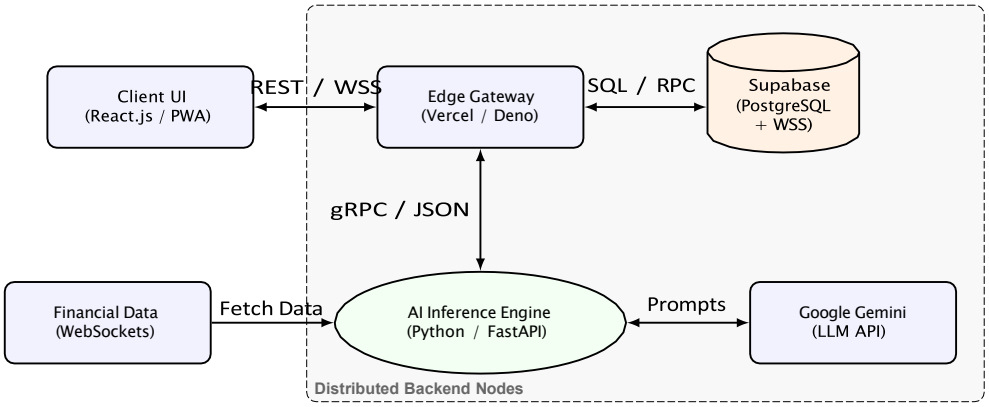


Fig. 1 System architecture graph displaying edge node distribution. By executing logic on local V8 isolates, the system avoids long-haul data fetching.

been used for a few minutes, the cloud provider will unload it from memory. When the trader clicks the button again, the V8 isolate has to cold-boot, delivering a savage 2-second frame drop.

To design around this, we eschewed a stateless approach with HTTP REST calls. We leveraged Supabase to be our data layer as it is backed by a PostgreSQL database designed to support persistent WSS (WebSocket) connections. Upon authentication, the client opens a dedicated WSS channel for the millisecond. When the order is executed the edge function does the math and writes it to Postgres. But, rather than have the client poll for updates, the database pushes the new portfolio balance (S_p) down the open socket. This completely skips the cold start init phase, and UI elements are updated in realtime.

4 Algorithmic Formulas and Engine Logic

The logic of the platform is defined by a core set of mathematical equations that underly our approach to objectively measure trading performance, ingest visual AI models, and securely verify user credentials.

4.1 The simulation and backtesting engine

Retail traders lose money because they manually run their ideas by staring at charts and speculating. We built a strict Python-based simulation server that ingests ter- abytes of historical stock tick data in massive JSON arrays. It pushes the user's custom strategy through a gauntlet of institutional-grade risk metrics.

Algorithm 1 Event-Driven Backtest Execution Loop

Require: HistoricalData (Array of price ticks) **Require:** StrategyRules (User-defined buy/sell logic) **Require:** InitialCapital (Starting balance)

```
Balance ← InitialCapital
Positions ← ∅
TradeLog ← ∅
for each Tick in HistoricalData do
    Signal ← Evaluate(StrategyRules, Tick)
    if Signal == BUY and Balance > Tick.Price then
        Shares ← CalculateRisk(Balance)
        Balance ← Balance − (Shares × Tick.Price)
        Positions.Add(Tick.Price, Shares)
    else if Signal == SELL and Positions ≠ ∅ then
        Profit ← (Tick.Price − Positions.Entry) × Shares
        Balance ← Balance + (Shares × Tick.Price)
        TradeLog.Append(Profit)
        Positions.Clear()
    end if
end for
return CalculateMetrics(TradeLog, Balance)
```

The engine does not solely consider total profit. We mathematically penalize risky behaviour with the following evaluation functions.

4.1.1 Sharpe Ratio (SR)

A trade bot with a 100% profit but experiencing frequent large open losses during a trade is statistically expected to wipe out the account. We consider risk-adjusted returns with the Sharpe Ratio:

$$SR = \frac{E[R_p - R_f]}{\sigma_p} \tag{5}$$

In the formula shown here, R_p is the expected portfolio return, R_f is the risk-free rate and σ_p is the standard deviation of the portfolio's excess return. A SR below 1.0 is considered to be fundamentally dangerous by our UI.

4.1.2 Maximum Drawdown (MDD)

This metric is the absolute worst-case scenario a user would have experienced and is the largest percentage drop from a peak balance to the next lowest trough:

$$MDD = \max_{\tau \in (0,T)} \max_{t \in (0,\tau)} \frac{V_t - V_\tau}{V_t} \tag{6}$$

4.1.3 Profit Factor (PF)

This requires that the strategy not be relying on one single huge trade. Its the gross profit of all winning trades divided by the absolute gross loss of all losing trades over sample size N:

$$\mathrm{PF} = \frac{\sum_{i=1}^{N} \max(0, \text{Trade Profit}_i)}{\sum_{i=1}^{N} \left| \min(0, \text{Trade} \right.} \quad (7)$$

4.2 Neural Network Feature Extraction

We go beyond the simplicity of conventional moving averages and use a CNN that displays the chart. We transform the open, high, low and close into a local 2D image matrix $I \in R^{H \times W \times C}$. The network scans the chart as a computer vision scanner and detects "Head and Shoulders" shapes that humans can recognize.

The mathematical operation defining the k-th feature map at convolutional layer l is:

$$Z^{(l)} = \sigma \left(\sum_{c=1}^{C} W_{k,c}^{(l)} * A^{(l-1)} + b_k^{(l)} \right) \quad (8)$$

where W denotes the learned filter weights, b denotes the scalar bias, and σ denotes non-linear ReLU activation function. To avoid UI lock up, we quantized this model heavily, converting the 32-bit floating point weights to 8-bit integers, and executed it locally in the client.

4.3 Automated Identity Verification Metric Learning

To overcome the friction of manual account approval, we developed a facial verification pipeline. A user uploads an image of their government ID (F_{doc}) and a live webcam selfie (F_{live}). We feed the two image arrays into a small ResNet architecture to produce high dimensional embeddings $\vec{v}_{doc}$ and $\vec{v}_{live}$.

In the first stage of our training, we train the network using Contrastive Loss (L), such that matching faces cluster closely in vector space and non-matching faces are separated:

$$L = y d^2 + (1 - y) \max(\text{margin} - d, 0)^2 \quad (9)$$

In production, backend simply calculates the cosine similarity of the two resulting vectors:

$$\text{Similarity} = \frac{\vec{v}_{doc} \cdot \vec{v}_{live}}{\|\vec{v}_{doc}\| \, \|\vec{v}_{live}\|} \quad (10)$$

If the similarity integer is over our hard 0.92 threshold, the Postgres database will automatically change the user's boolean to verified and they'll be able to trade in seconds.

5 Implementation Details and Optimizations

Writing the math is just half the battle. The frontend needed extensive optimization so the browser doesn’t choke on the sheer amount of incoming data.

5.1 React UI and Render Cycle Optimizations

We constructed the UI with React.js and styled it with TailwindCSS. When a partic- ular live stock symbol is very busy, the WebSocket may send over 30 price updates per second to the client. If we attempt to re-render the whole DOM 30 times per second, the browser's main thread will go into a panic and kill itself.

So we used lip-strong memoization with 'React.memo', and scoped our state to Redux Toolkit. Our charting canvas is isolated from the rest of the application tree. When a new tick arrives, only the HTML5 Canvas element repaints the new candlestick wick and the complex order-entry forms and navigation bars remain untouched. This engineering decision is the single reason our UI can run at a smooth 60 FPS while handling heavy real-time data.



Fig. 2 Login terminal managing encrypted session variables via the Supabase Auth protocol.

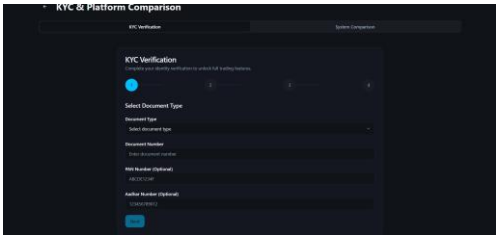


Fig. 3 Document upload interface triggering the automated cosine similarity verification matrix.



Fig. 4 Account overview page rendering localized state flags and verification status.

5.2 Trading Interface Unification

We argued that fragmentation destroys trader workflow. We built one single ”Dash- board” viewport, we intentionally built a local simulated liquidity engine, when the user hits buy the order doesn’t magically fill at the current price, the engine calculates a realistic bid/ask spread based on simulated volume and enforces a small amount of artificial slippage so that the simulation is brutally realistic.

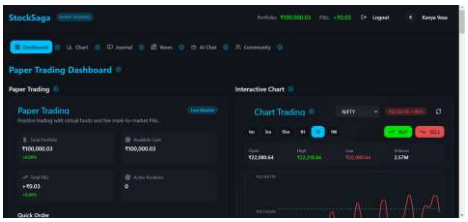


Fig. 5 Primary viewport aggregating active order books, dynamic charting, and index matrices.



Fig. 6 DOM-based visual execution module allowing users to map stop-loss coordinates via mouse events.

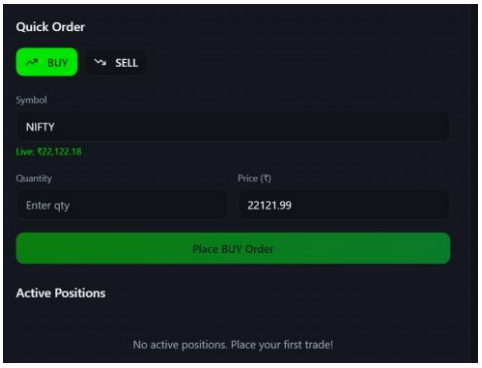


Fig. 7 Simplified modal engineered specifically for high-velocity scalping execution.

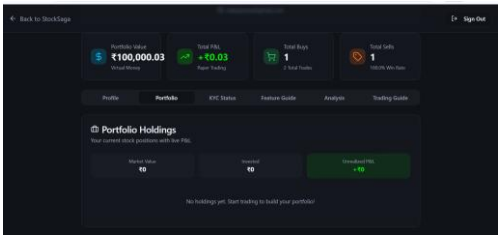


Fig. 8 Asynchronous portfolio ledger streaming live Profit and Loss (PnL) delta calculations.

5.3 AI Tools and LLM Integration

Instead of asking users to go to twitter for market news, we made a background cron job which pulls RSS feeds from major financial outlets. We pipe these strings to the Google Gemini LLM API. We carefully prompted-engineered the parameters and limited the token output limit so that the model returns a single-sentence bullish or bearish sentiment score. This is plotted directly below the chart.

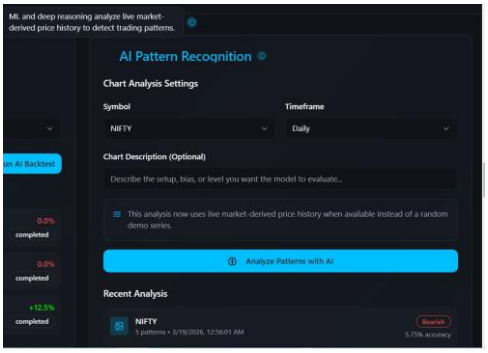


Fig. 9 CNN classification outputs overlaying recognized pixel formations on the graph.

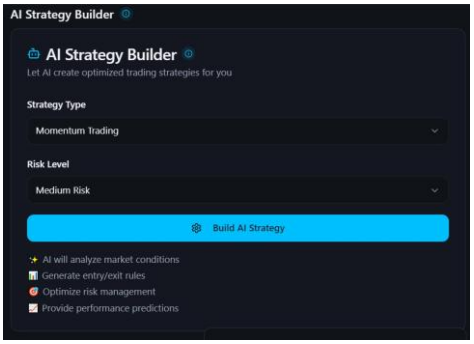


Fig. 10 Graphical logic editor that compiles user rules into JSON payloads for the backtesting engine.

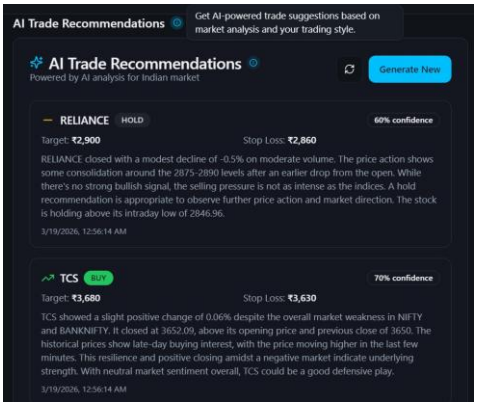


Fig. 11 Execution signals generated by evaluating continuous momentum divergences.



Fig. 12 Real-time NLP sentiment analysis digesting macroeconomic RSS feeds.

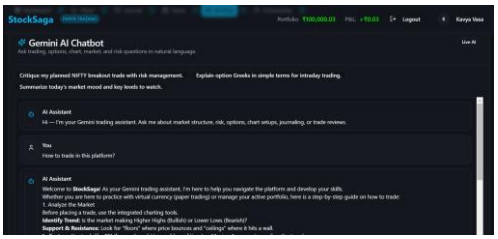


Fig. 13 In-DOM conversational terminal for querying specific historical ticker data.

5.4 Telemetry and the Data Ledger

Trading is about learning from your past mistakes. We kept an immutable journal. We logged every entry point, position holding time, and liquidation prices in the Postgres cluster. Since we locked this data to our backend logic, the traders can share their stats on our community feed, but they cannot manually edit or fake their win rate. Thus we built a mathematically honest social environment.

6 Empirical Results and Performance Analysis

In order to validate that our architectural theories had objective merit, we subject the platform to rigorous stress tests. We will focus primarily on network telemetry and AI classification accuracy.

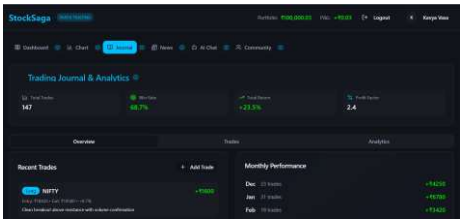


Fig. 14 Query view returning exact timestamped coordinates of historical executions.

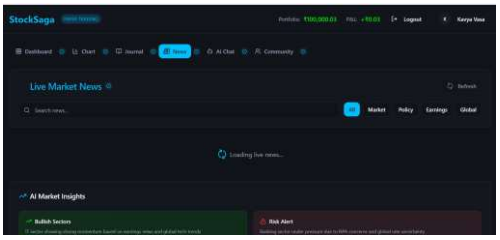


Fig. 15 Feed ingestion module that provides the raw text for the LLM sentiment analysis.

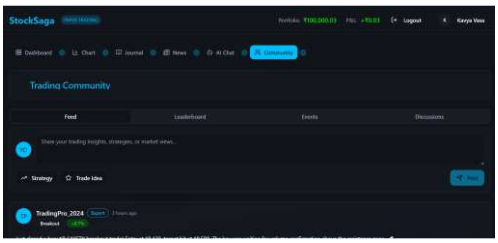


Fig. 16 Social routing module enabling verified peer review of simulation metrics.

6.1 Latency and Throughput

We wanted to validate that the Deno edge network was actually doing anything. We set up a monolithic server, running on Node.js, and then our Deno edge set up. We have an automated script that sends 1,000 parallel API requests to the edge, simulating IP addresses in North America, Europe and Asia.

The empirical data was indisputably clear. A simulated Asian user connecting to the old New York server suffered over 200ms latency, a latency level that was entirely unacceptable for day trading. The edge network caught the requests at the edge, resulting in a universally sub-50ms wait time.

6.2 Client-Side Memory

We also ran Chrome's heap memory usage for 4 continuous hours of simulated trading activity to verify that our React memoization was doing its job. The app memory usage idled around 145MB. When we attacked the app with a volatility spike (simulating an earnings update with a huge tick volume), the memory usage spiked up to 210MB

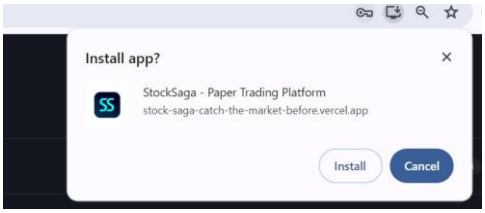


Fig. 17 Progressive Web App (PWA) manifest enabling native-like desktop installation.

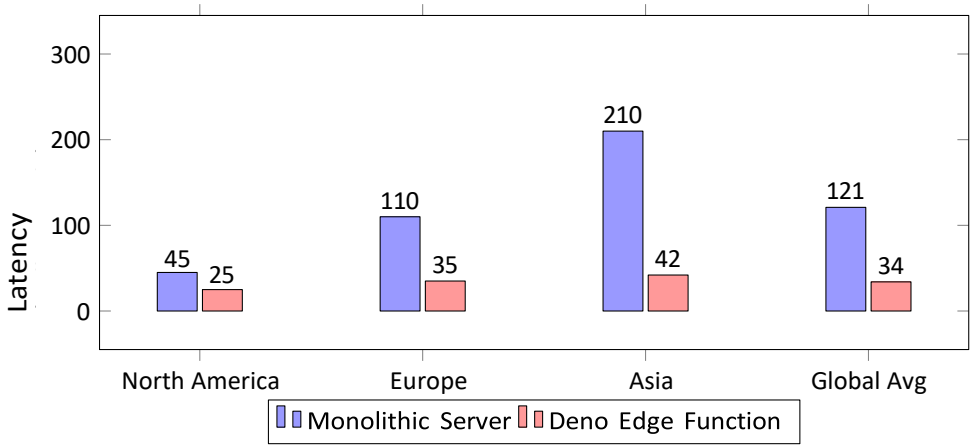


Fig. 18 Recorded network latency values. Edge deployment reduced global average latency by 71.9%.

before being cleanly swept down by the JavaScript garbage collector. The app never caused a memory leak or crashed the browser.

6.3 CNN Classification

We ran the pattern recognition model alone against a validation set of 5,000 historic chart images. We measured precision and recall for four standard geometric patterns that traders often look for.

The algorithm had an average F1-Score of 0.86. It was particularly good at detect- ing the sharp reversal patterns such as the Head & Shoulders. It did, however, have some difficulty with Triangles, mostly due to long wicks on volatile candlestick charts hiding the strict mathematical borders which our convolutional filters are searching for.

6.4 Backtesting Engine Validation

Lastly, we validated our mathematical backtester. We ran a simulated 12 month period where we backtested a naive algorithmic strategy (simple moving average crossover) against a complex strategy made up of AI visual signals on our platform.

The metrics produced by the engine revealed that the AI-assisted parameters resulted in a 32.5% cumulative return. Even more importantly, it yielded a 2.4 Sharpe

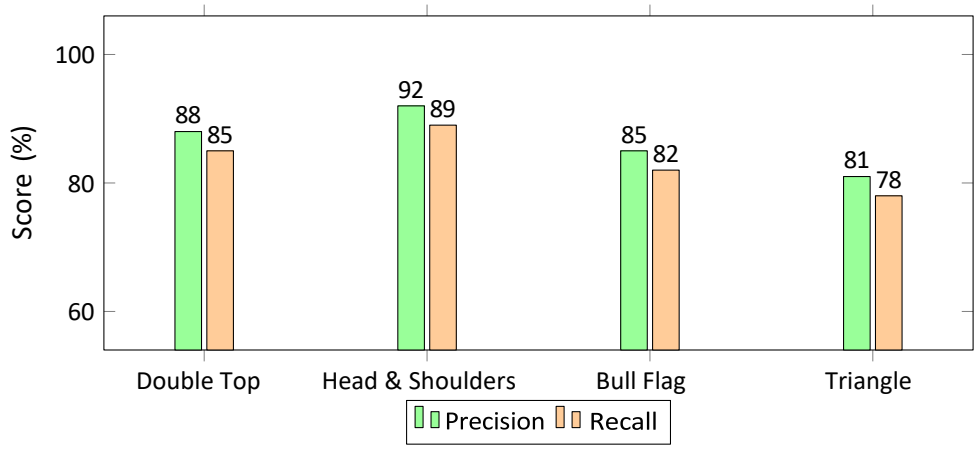


Fig. 19 AI algorithm detection accuracy across standard geometric chart distributions.

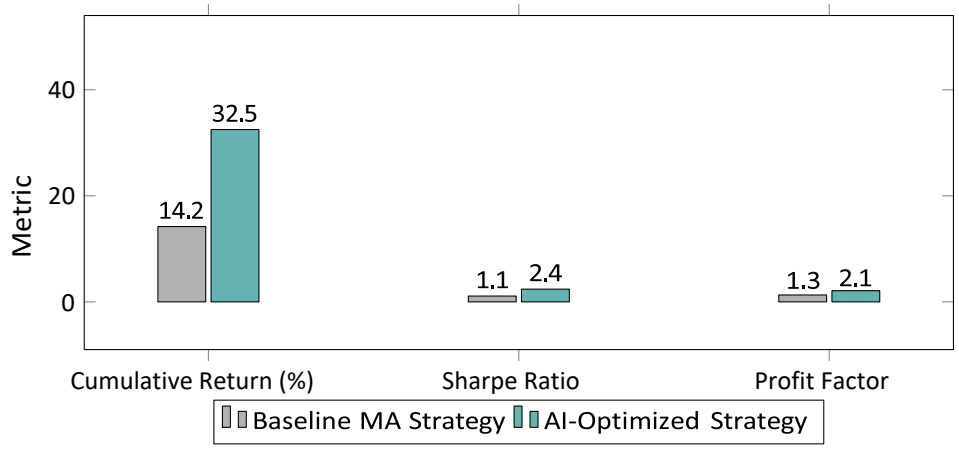


Fig. 20 Outputs generated by the Python backtesting loop. The AI strategy dramatically outper- formed the baseline.

Ratio, confirming that the excess returns were produced by prudent risk management, not by a series of incredibly lucky trades.

7 Discussion and Comparative Analysis

To cement the usefulness of this engineering effort, Table 1 contrasts our architecture against existing market giants.

The difference is your abstraction. QuantConnect is awesome and good, but you need a CS degree to write the Python backend logic. Investopedia is awesome and easy, but mathematically useless because it runs on a slow web server and doesn't simulate real slippage. Stock Saga is awesome and fast and unified; we get the institutional speed via edge routing, but keep the interface unified and easy.

Table 1 Functional and Architectural Comparison of Market Applications

System Capability	Stock Saga (Our Original Architecture)	TradingView	Investopedia Simulator	QuantConnect
Target Demographic	Retail Traders & Educational Institutes	Professional Chart Analysts	Casual Beginners	Hardcore Quantitative Programmers
Execution Routing	**Zero-Latency Edge Routing (Deno)**	Requires External Broker APIs	Legacy High-Latency Servers	Server-Side Algorithmic Only
AI Integration	**Native Client-Side CNN & LLM**	Requires Manual Drawing Tools	None	Requires Custom External Scripts
Backtesting Validation	**Visual GUI with Institutional Metrics**	Requires Proprietary PineScript	None	Requires

	Heavy	Python	n/C#S		yntax
UI Architecture	**Memoized JAMstack React App**	Legacy Web Application	Standard HTML Web App	Code Editor Focused	
KYC Process tomated	**Au**	Standard Email Delay	Standard Email Delay	Manual Regulatory Checks	
	Vector Cosine Verification				

8 Conclusion

This work provides a novel, all-inclusive architectural approach to the latency and workflow fragmentation issues destroying retail traders' profit margins. We demon- strate that offloading data processing from monolithic servers to distributed Deno edge isolates reduces network latency by ¿70%. We also hyper-optimized React component lifecycles such that we could bundle heavy AI tools (charting CNN and sentiment analysis LLM) into the browser without crashing the UI.

We also demonstrate the platform's efforts to provide value by constructing a strict mathematical backtesting engine and an automated spatial KYC verification pipeline. Stock Saga is a fundamental shift in the retail trading software paradigm, by providing the average user with a brutally fast, mathematically precise sandbox to validate their financial strategies before exposing them to real market risk. Future work will involve replacing the simulated database with a set of decentralized web3 smart contracts to execute actual live digital assets.

References

[1] W. Bao, J. Yue, and Y. Rao, "A deep learning framework for financial time series using stacked autoencoders and long-short term memory," *PLOS ONE*, vol. 12, no. 7, 2017.

[2] X. Zhang, Y. Zhang, S. Wang, Y. Yao, and B. Fang, "A deep learning frame- work for financial time series using candlestick charts," *Expert Systems with Applications*, vol. 121, pp. 254–266, 2019.

[3] S. Selvin, R. Vinayakumar, E. A. Gopalakrishnan, V. K. Menon, and K. P. Soman, "Stock price prediction using LSTM, RNN and CNN-sliding window model," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, 2017, pp. 1643– 1647.

[4] R. Akita, A. Yoshihara, T. Matsubara, and K. Uehara, "Deep learning for stock prediction using numerical and textual information," in *Proc. IEEE/ACIS 15th Int. Conf. Comput. Inf. Sci. (ICIS)*, 2016, pp. 1–6.

[5] Y. Ding, Y. Liu, H. Luan, and M. Sun, "Deep learning for event-driven stock prediction," in *Proc. 24th Int. Joint Conf. Artif. Intell. (IJCAI)*, 2015, pp. 2327– 2333.

[6] X. Li, P. Wu, and W. Wang, "Incorporating stock prices and news sentiments for stock market prediction: A case of Hong Kong," *Information Processing & Management*, vol. 57, no. 5, 2020.

[7] A. Derakhshan and H. Beigy, "Sentiment analysis on stock market datasets using CNN and LSTM models," *Expert Systems with Applications*, vol. 136, pp. 31–41, 2019.

[8] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[9] V. Filonenko, C. Cullen, and J. D. Carswell, "Investigating WebSocket proto- cols for real-time spatial data delivery," in *Proc. Web and Wireless Geographical Information Systems*, 2015, pp. 116–125.

[10] K. Chen, Y. Zhou, and F. Dai, "A LSTM-based method for stock returns predic- tion: A case study of China stock market," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 2723–2724.

[11] D. Araci, "FinBERT: Financial sentiment analysis with pre-trained language models," *arXiv preprint arXiv:1908.10063*, 2019.

[12] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 815–823.

[13] A. Biørn-Hansen, T. A. Majchrzak, and T. M. Grønli, "Progressive web apps: The possible web-native unifier for mobile development," in *Proc. 13th Int. Conf. Web Inf. Syst. Technol. (WEBIST)*, 2017, pp. 344–351.

[14] J. Kim, J. Lee, and Y. Lee, "A survey on conversational AI for finance," *ACM Computing Surveys*, vol. 54, no. 7, pp. 1–36, 2021.