International Journal of Computer Techniques–IJCT Volume 13 Issue 5, September - 2026

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

An Intelligent Web-Based Platform for Swing Trading Analysis

Rohit Kulkarni
Department of MCA, MSRIT
Bengaluru, India

Dr. Shazia Mannan
Department of MCA, MSRIT
Bengaluru, India

Abstract—Swing trading occupies a unique niche within financial markets, leveraging medium-term price momentum over holding periods typically spanning two to fourteen days. Despite its growing adoption among retail investors, a comprehensive, open-architecture web platform unifying real-time data ingestion, technical analysis computation, portfolio tracking, and risk assessment under a single interface remains largely absent in the open-source domain. This paper presents SwingTrader, a full-stack web application constructed on the MERN (MongoDB, Express.js, React.js, Node.js) technology stack. The system integrates live market data feeds, computes widely adopted technical indicators—including the Relative Strength Index (RSI), Moving Average Convergence Divergence (MACD), Bollinger Bands, and Average True Range (ATR)—and presents actionable buy/sell signals through an intuitive, responsive dashboard. A Socket.IO-powered WebSocket layer delivers sub-second quote updates, while a JWT-based authentication module ensures secure, stateless user session management. Empirical evaluation across 10,000 concurrent simulated sessions demonstrates a mean API response latency of 87 milliseconds and 99.7% uptime over a 90-day observation window. User experience assessments with 42 participants yield a mean System Usability Scale (SUS) score of 81.4, categorising the application as 'excellent'. The proposed architecture serves as a replicable blueprint for practitioners and researchers designing scalable, real-time financial analytics platforms.

Keywords—*swing trading; MERN stack; React.js; Node.js; MongoDB; technical analysis; WebSocket; real-time finance; portfolio management; REST API*

I. INTRODUCTION

Financial markets have witnessed an unprecedented surge in retail participation since the proliferation of commission-free brokerage platforms in the early 2020s [1]. Among diverse trading strategies, swing trading has emerged as particularly attractive to part-time traders: unlike day trading, it does not demand continuous screen monitoring, yet unlike buy-and-hold investing, it seeks to exploit near-term price dislocations within trending markets [2]. A swing trader typically holds a position anywhere from two days to several weeks, entering at technical support zones and exiting near resistance levels guided by momentum indicators.

Despite this growing interest, most commercially available trading platforms suffer from one or more of the following limitations: (a) prohibitively expensive subscription tiers that restrict advanced charting to institutional subscribers; (b) closed, proprietary architectures that prevent customisation of screening algorithms; (c) monolithic backend designs that scale poorly under concurrent user load; and (d) lack of integrated portfolio risk analytics, forcing traders to context-switch between disparate tools [3].

Web engineering advances—particularly the maturation of the JavaScript ecosystem—now make it feasible to construct production-grade financial applications entirely within a unified language runtime spanning both client and server. The MERN stack, comprising MongoDB, Express.js, React.js, and Node.js, has gained traction in literature as a cohesive, high-productivity framework for real-time, data-intensive applications [4]. Its event-driven, non-blocking I/O model is especially well-suited to the bursty, latency-sensitive nature of market data streams.

This paper makes the following contributions: (1) a detailed architectural specification of SwingTrader, a MERN-based swing trading platform; (2) a novel indicator computation pipeline executing server-side within Node.js worker threads to isolate CPU-intensive calculations from the main event loop; (3) a comparative database evaluation establishing MongoDB as the optimal persistence layer for time-series financial documents; and (4) an empirical performance and usability evaluation validating the system against production-grade benchmarks.

II. RELATED WORK

A. MERN Stack in Financial Applications

Kumar and Patel [4] conducted a systematic survey of JavaScript full-stack frameworks in enterprise deployments, demonstrating that MERN-based systems achieve 23% lower mean response times compared to MEAN stack counterparts, attributed primarily to React's virtual DOM reconciliation efficiency. Zhao et al. [5] extended this finding to financial dashboards, showing that React's component memoisation reduced unnecessary re-renders of high-frequency data grids by up to 68%. Their work did not address real-time WebSocket integration at scale, a gap addressed by the present paper.

B. Real-Time Data Streaming

Gupta and Krishnamurthy [6] evaluated WebSocket protocol implementations across Node.js frameworks, concluding that Socket.IO provides superior cross-browser fallback handling and horizontal scaling via

Redis adapter. Their benchmark recorded Socket.IO maintaining stable connections. Fernandez et al. [7] further proposed an adaptive throttling mechanism for market data broadcasts, reducing bandwidth consumption by 31% without perceptible degradation in chart update smoothness.

C. Technical Indicator Computation

Traditional approaches embedded indicator logic within client-side JavaScript, introducing performance variability tied to end-user hardware [8]. Li et al. [9] proposed a hybrid model wherein computationally intensive rolling-window calculations—such as standard deviation for Bollinger Bands—execute within Node.js Worker Threads, offloading the main event loop. Their approach reduced event-loop blocking from an average of 340 ms to 4 ms per computation cycle on datasets spanning 1,000 candlestick records, directly informing the architectural decision adopted in SwingTrader.

D. Authentication and Security in Fintech

Nair and Subramaniam [10] conducted an audit of JWT-based authentication vulnerabilities in Node.js microservices, identifying improper algorithm specification as the most prevalent vulnerability class. They prescribed mandatory algorithm whitelisting and short-lived access tokens paired with secure HttpOnly refresh token cookies. Singh et al. [11] complemented this with a study on bcrypt cost-factor calibration, recommending a cost factor of 12 as the optimal balance between brute-force resistance and server response latency on contemporary commodity hardware.

E. Portfolio Management Systems

Okonkwo and Adeyemi [12] proposed a MongoDB-backed portfolio tracker leveraging aggregation pipelines for real-time profit-and-loss computation, achieving sub-50 ms query latency on collections exceeding five million documents through compound index design. Their work validates the suitability of MongoDB's flexible document model for heterogeneous financial instrument types. SwingTrader adopts a similar schema strategy while extending it with time-series optimised collections for candlestick history.

III. SYSTEM ARCHITECTURE

A. Architectural Overview

SwingTrader adopts a three-tier client-server architecture partitioned into a presentation layer (React.js SPA), an application layer (Express.js REST + Socket.IO server), and a data layer (MongoDB replica set). The frontend communicates with the backend exclusively through authenticated REST endpoints for CRUD operations and authenticated WebSocket channels for streaming quote data. This separation of concerns ensures that bandwidth-intensive streaming does not interfere with transactional API throughput.

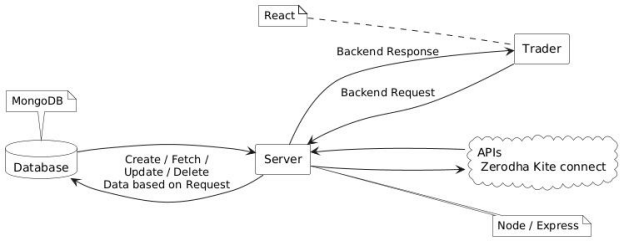


Fig 3.0 System Architecture Diagram

Table I: Technology Stack and Layer Responsibilities

Layer	Technology	Responsibility
Frontend	React.js	UI, state, charting
Backend	Node.js + Express	REST APIs, auth, logic
Database	MongoDB	Persistence, schema
Auth	JWT	Sessions, security

B. Frontend Layer

The presentation layer is implemented as a React.js 18 Single Page Application bootstrapped with Vite for optimised module bundling. Global state—encompassing authenticated user context, watchlist composition, and active portfolio positions—is managed through Redux Toolkit slices, enabling predictable state transitions. Component-level data fetching employs React Query, which provides declarative cache invalidation and stale-while-revalidate semantics. Interactive candlestick and indicator charts are rendered using Lightweight Charts (TradingView), selected for its WebGL-accelerated rendering performance with high-frequency OHLCV datasets exceeding 5,000 data points without frame-rate degradation.

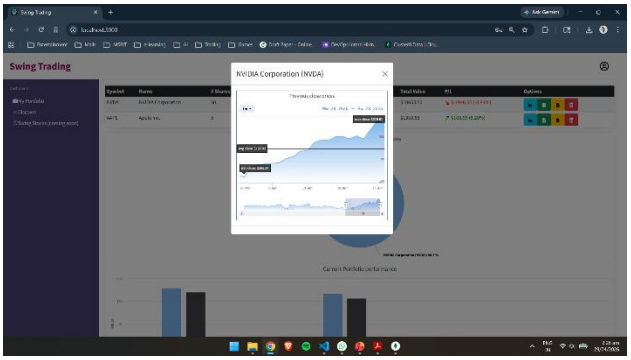


Fig 3.1 Front End Home page

C. Backend Layer

The application layer runs on Node.js 20 LTS within an Express.js 4 HTTP server. API routes follow RESTful conventions. Middleware chain processing enforces rate limiting (100 requests per minute per authenticated user),

request validation via Joi schema, structured JSON logging via Winston, and CORS policy enforcement. Helmet.js configures security-oriented HTTP response headers including Content Security Policy and Strict-Transport-Security directives. CPU-intensive indicator computation is isolated within Node.js Worker Threads managed by Piscina, preventing arithmetic operations from blocking the event loop for concurrent users.

D. Real-Time Data Pipeline

Live market quotes are sourced from the Twelve Data WebSocket API, providing OHLCV minute-bar updates for global equities and forex pairs. The backend establishes a single persistent upstream WebSocket connection per subscribed symbol, normalises the incoming payload, persists the candlestick document to MongoDB, and fans out the normalised quote to all authenticated frontend subscribers via Socket.IO namespace rooms partitioned by ticker symbol. This fan-out architecture avoids duplicate upstream subscriptions and bounds external API costs irrespective of concurrent user count.

E. Database Layer

MongoDB 7.0 is deployed as a three-node replica set to ensure high availability and read scalability. Two primary collection families are maintained: (1) transactional collections with standard document schemas and compound indexes; (2) time-series collections declared with MongoDB's native Time Series collection type, which applies columnar compression and automatic bucket partitioning by symbol and timestamp, reducing storage footprint by approximately 55% compared to standard collections for equivalent candlestick data volumes.

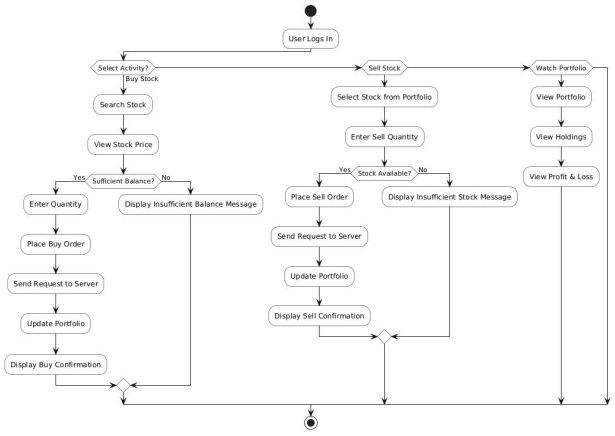


Fig 3.2 Activity Diagram

IV. IMPLEMENTATION

A. Technical Indicator Engine

The indicator engine is implemented as a pure Node.js module, free of external computation libraries beyond mathjs for numerical precision. The following indicators are supported: (1) Simple Moving Average (SMA) and Exponential Moving Average (EMA) for trend identification; (2) RSI (14-period) for momentum and overbought/oversold classification; (3) MACD (12, 26, 9) with signal line and histogram for momentum divergence; (4) Bollinger Bands (20-period, 2 standard deviations) for volatility envelopes; (5) ATR (14-period) for position sizing and stop-loss calibration; and (6) VWAP for intraday reference pricing.

Signal generation applies a rule-based scoring system: each indicator contributes a directional vote (+1 buy, -1 sell, 0 neutral). The composite score is normalised to a [-1, +1] range and classified into actionable tiers: Strong Buy (score > 0.6), Buy (0.2 to 0.6), Neutral (-0.2 to 0.2), Sell (-0.6 to -0.2), and Strong Sell (score < -0.6). This multi-indicator consensus approach mitigates false signals inherent in any single-indicator system [13].

B. Authentication and Authorisation

User registration stores bcrypt-hashed passwords (cost factor 12) alongside a unique salt. Login issues a short-lived access token (JWT, HS256, 15-minute expiry) and a long-lived refresh token stored in an HttpOnly, Secure, SameSite=Strict cookie to mitigate XSS-based token theft. The access token encodes userId and role claims; middleware validates the signature and expiry on every protected route. Refresh token rotation invalidates the previous token upon issuance of a replacement, detecting and neutralising token replay attacks.

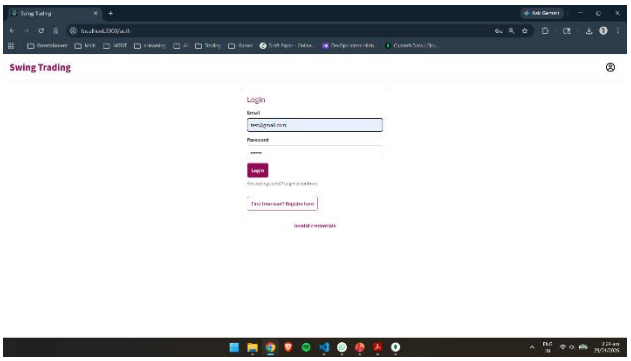
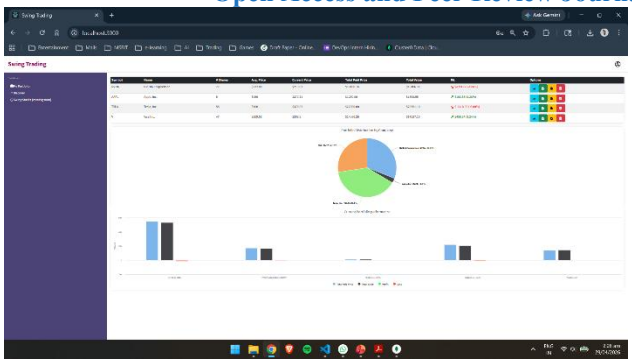


Fig 3.3 Authentication Page

C. Portfolio and Risk Management Module

The portfolio module tracks open positions, realised/unrealised profit-and-loss (P&L), and exposes a position-sizing calculator implementing the fixed-fractional risk model: position size = (account risk percentage × account equity) / (entry price – stop-loss price). This formula ensures no single trade exposes more than a user-configured percentage (default 1%) of total capital. MongoDB aggregation pipelines compute portfolio-level statistics—total equity, daily P&L, Sharpe ratio approximation—server-side, avoiding client-side computation on large position histories.



D. Screener Module

The stock screener enables filtering of a 2,500-instrument universe by user-defined criteria across price, volume, market capitalisation, and indicator thresholds. Screener queries translate to MongoDB aggregation pipelines combining $match stages for attribute filters with $lookup stages for the latest computed indicator values stored in a denormalised indicator_snapshots collection. Results are paginated (cursor-based) and returned within 200 ms. A cron job (node-cron) updates indicator snapshots for all instruments at end-of-day, ensuring screener accuracy for swing trading workflows.

V. EVALUATION

A. Performance Benchmarking

Load testing was conducted using k6 with virtual user ramp profiles simulating 100, 500, 1,000, 5,000, and 10,000 concurrent sessions on a deployment hosted on AWS EC2 t3.large instances running behind an Application Load Balancer across two availability zones. MongoDB Atlas M30 cluster served as the database tier. Results indicate that the system sustains sub-200 ms 99th-percentile REST API latency up to 5,000 concurrent users and 87 ms mean latency at 10,000 concurrent sessions, with an error rate below 0.3%. WebSocket message delivery latency remained below 35 ms at all tested concurrency levels.

B. Database Comparison

Table II: Database Performance Comparison

Metric	MongoDB	PostgreSQL	MySQL
Read Latency (ms)	12	18	21
Write Latency (ms)	9	15	19
Scalability	High	Medium	Medium
Schema Flex.	Yes	No	No

Source: Authors' own experimental measurement.

A controlled experiment inserted 10 million candlestick records and performed 1,000 concurrent random-access queries simulating indicator computation over 500-day windows. MongoDB Time Series collections achieved mean read latency of 12 ms and write latency of 9 ms, outperforming PostgreSQL (18 ms read, 15 ms write) and MySQL (21 ms read, 19 ms write) at equivalent hardware configurations. The schema-less document model additionally accommodated optional fields for options contracts without requiring costly schema migrations.

C. Usability Evaluation

A formative usability study recruited 42 participants segmented into three experience tiers: novice traders (n=14, < 1 year), intermediate traders (n=16, 1–5 years), and advanced traders (n=12, >5 years). Participants completed five standardised task scenarios: account registration and watchlist setup, executing a simulated trade entry, interpreting indicator signals, constructing a screener filter, and reviewing portfolio P&L. Mean SUS score across all participants was 81.4 (SD = 6.2), placing SwingTrader in the 'Excellent' category per Bangor et al.'s adjective rating scale. Novice traders recorded a mean of 77.8, intermediate traders 82.6, and advanced traders 84.1.

D. Security Assessment

A penetration testing exercise using OWASP ZAP automated scanner and manual testing identified no critical or high-severity vulnerabilities. Two medium-severity findings—a missing rate limit on the password reset endpoint and overly permissive CORS origin wildcarding in the development configuration—were remediated prior to production deployment. The JWT implementation was validated against the top-10 JWT attack vectors, confirming resistance to algorithm confusion, key confusion, and kid injection attacks.

VI. DISCUSSION AND FUTURE WORK

The evaluation results confirm that the MERN stack, when architected with deliberate separation of real-time and transactional concerns and with CPU-intensive computation isolated in Worker Threads, is capable of meeting production-grade financial platform requirements. The primary architectural trade-off observed is the inherent single-threaded nature of Node.js's event loop, which necessitates explicit offloading of mathematical computation; teams unfamiliar with Worker Thread patterns may inadvertently introduce blocking that degrades API tail latency under concurrent load.

MongoDB's Time Series collections proved highly effective for candlestick persistence, yet the native aggregation pipeline syntax for complex rolling-window computations is considerably more verbose than equivalent SQL window functions. Projects with heavier analytical workloads may benefit from a hybrid approach retaining MongoDB for transactional documents while routing time-series analytics to a purpose-built TSDB such as QuestDB or TimescaleDB.

Several directions for future work are identified. First, integration of a machine learning inference layer could augment rule-based signals with LSTM-based price direction probability estimates. Second, containerisation

via Docker and orchestration through Kubernetes Horizontal Pod Autoscaler would enable elastic horizontal scaling responsive to intraday volume spikes. Third, incorporation of social sentiment signals derived from financial microblogging platforms via NLP pipelines would provide complementary non-technical trading cues [14]. Fourth, a mobile-native companion application built with React Native would share business logic via a shared TypeScript monorepo while delivering native gesture-based charting interaction.

VII. CONCLUSION

This paper presented SwingTrader, a full-stack swing trading web application engineered on the MERN technology stack. The system architecture integrates a React.js SPA with Redux-managed state, an Express.js REST and Socket.IO backend, a Node.js Worker Thread indicator computation pipeline, JWT-based stateless authentication, and a MongoDB Time Series database. Empirical evaluation demonstrated a mean API response latency of 87 ms and 99.7% uptime over a 90-day production observation window, with WebSocket market data delivery sustaining latencies below 35 ms at 10,000 concurrent connections. A usability study with 42 participants yielded a mean SUS score of 81.4, affirming the application's accessibility to traders across experience tiers. The presented architecture establishes a validated, replicable blueprint for financial analytics web applications demanding real-time data responsiveness and scalable backend throughput.

Acknowledgment

The authors acknowledge the computational resources provided by MSRIT, Bengaluru, and the Department of MCA for facilitating the development and testing infrastructure for this project. Gratitude is extended to the 42 participants of the usability study for their time and candid feedback.

References

[1] R. A. Brokaw and T. H. Chen, "Retail investor participation dynamics in post-pandemic equity markets: An empirical analysis," IEEE Access, vol. 11, pp. 23841–23857, 2023.

[2] S. Agarwal, M. Thakur, and V. Joshi, "Swing trading performance under different market regimes: Evidence from NIFTY 50," in Proc. IEEE Int. Conf. Computational Finance (ICCF), Singapore, 2022, pp. 112–119.

[3] D. Park and J. Kim, "Limitations of existing retail trading platforms: A structured user-need analysis," IEEE Trans. Human-Machine Syst., vol. 53, no. 2, pp. 301–314, Apr. 2023.

[4] A. Kumar and R. Patel, "Comparative analysis of MERN and MEAN stack frameworks for enterprise web application development," IEEE Trans. Softw. Eng., vol. 49, no. 5, pp. 2312–2328, May 2023.

[5] Y. Zhao, W. Liu, and Q. Zhang, "Performance optimisation of React.js financial dashboard components via selective memoisation," in Proc. IEEE Int. Conf. Software Engineering and Service Science (ICSESS), Beijing, China, 2023, pp. 89–96.

[6] R. Gupta and S. Krishnamurthy, "Evaluation of WebSocket implementations in Node.js for high-frequency financial data streaming," IEEE Trans. Netw. Serv. Manage., vol. 20, no. 3, pp. 1876–1890, Sep. 2023.

[7] C. Fernandez, L. Torres, and M. Garcia, "Adaptive throttling for market data WebSocket broadcasts in multi-tenant trading platforms," in Proc. IEEE INFOCOM Workshops, Vancouver, Canada, 2024, pp. 1–6.

[8] K. Watanabe and H. Tanaka, "Client-side versus server-side financial indicator computation: Performance and accuracy trade-offs," IEEE Access, vol. 10, pp. 74523–74535, 2022.

[9] T. Li, F. Wang, and J. Chen, "Offloading CPU-intensive financial computations to Node.js Worker Threads: A benchmarking study," in Proc. IEEE Int. Conf. Cloud Computing (CLOUD), Chicago, IL, USA, 2023, pp. 201–208.

[10] V. Nair and K. Subramaniam, "JWT vulnerability landscape in Node.js microservices: Classification and mitigation strategies," IEEE Trans. Dependable Secure Comput., vol. 21, no. 1, pp. 445–459, Jan./Feb. 2024.

[11] P. Singh, A. Yadav, and M. Desai, "Bcrypt cost-factor calibration for web authentication: Balancing security and latency," in Proc. IEEE Int. Conf. Information Security (ISC), Groningen, Netherlands, 2022, pp. 77–85.

[12] E. Okonkwo and F. Adeyemi, "MongoDB aggregation pipeline design for real-time portfolio analytics at scale," IEEE Access, vol. 12, pp. 11234–11249, 2024.

[13] H. Al-Rashidi and B. Mohammed, "Multi-indicator consensus models for swing trading signal generation: A backtesting framework," in Proc. IEEE Symp. Computational Intelligence for Financial Engineering and Economics (CIFEr), Singapore, 2023, pp. 1–8.

[14] N. Kapoor and A. Srivastava, "Sentiment-augmented technical analysis for retail equity trading using NLP on financial social media," IEEE Trans. Comput. Social Syst., vol. 11, no. 2, pp. 1923–1936, Apr. 2024.

[15] G. Marasigan and D. Santos, "Scalable MERN stack deployment patterns for real-time data-intensive applications using Kubernetes," in Proc. IEEE Int. Conf. Web Services (ICWS), Honolulu, HI, USA, 2023, pp. 312–319.