International Journal of Science, Strategic Management and Technology
Volume **02** Issue **04** April-**2026** | ISSN: **3108-1762 (Online)** | Impact Factor: 3.8
An International, Peer-Reviewed, Open Access Scholarly Journal Indexed in recognized academic databases

A Scalable AI-Driven Natural Language Interface for Algorithmic Trading with Strategy Validation and Asynchronous Execution

Author Details:

Janhvi N. Patil[1], Durvesh H. Patil[2], Rushikesh K. Binnar[3], Chaitan B. Jadhav[4], Mrs. S. H. Adke[5], Mr. P. R. Pachorkar[6]

[1] Department of Information Technology, MVPS's KBTCOE, Nashik, India
[2] Department of Information Technology, MVPS's KBTCOE, Nashik, India
[3] Department of Information Technology, MVPS's KBTCOE, Nashik, India
[4]Department of Information Technology, MVPS's KBTCOE, Nashik, India
[5] Department of Information Technology, MVPS's KBTCOE, Nashik, India
[6] Department of Information Technology, MVPS's KBTCOE, Nashik, India
Corresponding Author: janhvinpatil@gmail.com | Department of IT, MVPS's KBTCOE, Nashik, India

https://doi.org/10.55041/ijsmt.v2i4.156

Cite this Article: Patil[1],, J. N., Patil,, D. H., Binnar, R. K., Jadhav, C. B., Adke, S. H. & Pachorkar, P. R. (2026). A Scalable AI-Driven Natural Language Interface for Algorithmic Trading with Strategy Validation and Asynchronous Execution. International Journal of Science, Strategic Management and Technology, *02*(04). https://doi.org/10.55041/ijsmt.v2i4.156

License: This article is published under the Creative Commons Attribution 4.0 International License (CC BY 4.0), permitting use,

Abstract—

This paper presents an advanced AI-assisted trading platform designed to enable users to conceptualize, test, and activate cryptocurrency trading strategies using plain-language descriptions, without requiring programming knowledge. Traditional algorithmic trading solutions impose significant technical barriers—demanding proficiency in coding and quantitative finance—that systematically exclude a broad segment of individual investors. The proposed system addresses this challenge by incorporating a Large Language Model (LLM) capable of interpreting free-form user descriptions and converting them into structured, machine-readable trading instructions. The platform is built on a distributed, service-oriented architecture comprising a Flutter-based mobile frontend, a Django REST API for core application logic, and a FastAPI microservice dedicated to LLM inference. Computationally intensive workflows—including strategy generation, simulation over historical data, and trade order dispatch—are handled through a non-blocking task pipeline orchestrated by Redis and Celery. Prior to execution, each strategy is passed through a rule-based validation module that verifies logical integrity and compliance with defined risk parameters. A built-in backtesting engine enables retrospective performance analysis, and live trading is supported through persistent connections to cryptocurrency exchange APIs. Experimental results demonstrate that the platform substantially reduces the skill threshold required for strategy development, expands market participation for non-technical users, and maintains reliable throughput through distributed task handling. The modular, loosely coupled design supports horizontal scaling and straightforward integration with production financial systems.

Keywords— Algorithmic Trading; Natural Language Processing; Large Language Models (LLM); Strategy Generation; Backtesting; Asynchronous Processing; Distributed Systems; Financial Technology; Celery; Redis; Django REST; FastAPI

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

I. INTRODUCTION

The proliferation of automated trading systems has fundamentally altered the dynamics of modern financial markets, enabling high-speed, rule-governed order execution driven by continuously evolving market signals [16]. Despite this growing adoption, developing effective trading strategies still demands a rare combination of technical development skills, statistical reasoning, and familiarity with how markets operate at a structural level [15]. For the vast majority of individual traders who possess strong market instincts but limited programming experience, this combination of prerequisites effectively prevents algorithmic participation [8].

Existing commercial trading platforms attempt to address this gap through visual rule builders or domain-specific scripting languages such as PineScript and Python-based SDKs [1]. Although these tools offer experienced users fine control over strategy logic, they still assume that the user can already reason algorithmically and translate intuition into formal code [8]. As a result, a significant portion of potential algorithmic traders remains locked out of these tools [16].

Recent advances in generative AI, and in LLMs particularly [5], have opened promising pathways for translating informal human intent into structured computational representations [4]. These models demonstrate strong capabilities in parsing unstructured text, recognizing domain-specific entities, and producing coherent structured outputs [4], [6]. However, their use in trading has largely been confined to price prediction or news sentiment analysis [3], and the problem of enabling full end-to-end strategy authorship from natural language remains largely unresolved.

This paper introduces a unified platform in which users describe their trading intentions in everyday language, and the system automatically translates those descriptions into validated, executable strategy logic. The platform combines an LLM-based generation component [5] with a deterministic validation layer [2] that enforces correctness and risk compliance. A simulation engine enables retrospective testing on historical price data [1], while an asynchronous task scheduler [10] ensures that resource-intensive computations do not interfere with user experience.

A distinctive contribution of this work is the novel application of LLMs in algorithmic strategy authorship—enabling traders to generate executable logic directly from conversational input. Rather than relying on manual coding or predefined configuration templates, the proposed system leverages generative AI to bridge the gap between intuitive trading reasoning and formal computational logic. This approach significantly lowers the barrier to strategy development and demonstrates how LLMs can automate complex financial system design in a user-accessible manner.

The primary contributions of this work are as follows: (i) a natural language interface for strategy creation that demands no programming knowledge [4], [5]; (ii) a constraint-driven validation framework that screens generated strategies for logical errors and risk violations before any execution occurs [2]; (iii) a distributed asynchronous task layer that handles concurrent workloads without incurring user-facing latency [7], [10]; (iv) a fully integrated pipeline from strategy description through simulation to live deployment, within a single coherent system [8]; and (v) a demonstration that LLMs can serve as a practical engine for natural language-driven strategy generation, reducing reliance on developer expertise [5].

II. LITERATURE REVIEW

Automated trading has matured substantially over the past decade, with numerous commercial platforms now providing strategy construction, simulation, and execution capabilities across equities, derivatives, and digital assets [16]. Platforms such as TradeTron, AlgoBulls, and QuantMan have gained adoption by offering pre-built rule templates and drag-and-drop strategy assemblers [8]. Despite their operational sophistication, these systems share a critical limitation: they require users to manually encode trading logic via graphical editors or proprietary scripting interfaces [1]. Traders without programming expertise or formal quantitative training find these platforms largely inaccessible for converting market knowledge into deployable strategies [15].

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

A distinct body of research has explored the application of statistical and neural learning methods in financial decision-making [2], [3]. Evolutionary algorithms, model-based reinforcement learning, and variants of convolutional or recurrent neural networks have been used to address challenges such as portfolio rebalancing, momentum signal identification, and execution optimization [3], [6]. While these techniques achieve competitive results within constrained experimental conditions, they primarily target the prediction and optimization layers of a trading pipeline, rather than simplifying the upstream process of strategy specification [3]. Their practical deployment is also complicated by extensive data requirements, hand-crafted feature engineering, and sensitivity to distribution shifts in live market data [6].

Progress in transformer-based language modeling [4], [5] has introduced qualitatively new capabilities for interpreting open-ended text and generating structured outputs. Within financial NLP, applications have emerged for sentiment scoring of earnings calls, summarization of regulatory filings, and information extraction from analyst reports [4]. However, the distinct challenge of converting a trader's informal description into a syntactically valid and risk-compliant algorithmic strategy has received limited systematic attention [5]. The few prototype systems that touch on this problem typically omit a formal validation step [2], making their outputs unsuitable for live deployment where erroneous logic could trigger unintended market orders.

A further limitation of existing work is its siloed architecture. Strategy specification, rule verification, historical simulation, and live execution are almost universally handled by separate, loosely connected tools [8], forcing practitioners to manage complex handoffs and manually reconcile inconsistencies. Computationally intensive steps such as multi-year parameter sweeps are frequently executed synchronously within single-threaded services [7], introducing latency that compounds under concurrent usage.

The proposed platform addresses this constellation of limitations through a purpose-built, fully integrated architecture. An LLM-powered module [5] converts informal user intent into structured strategy representations, immediately followed by a deterministic validation layer [2] that certifies logical soundness before any downstream processing is triggered. Asynchronous task execution [10], historical simulation [1], and direct broker connectivity [13], [14] are unified within a single extensible platform [8], delivering an end-to-end experience that is both accessible to non-technical users and robust enough for real-world deployment.

III. METHODOLOGY

A. System Overview

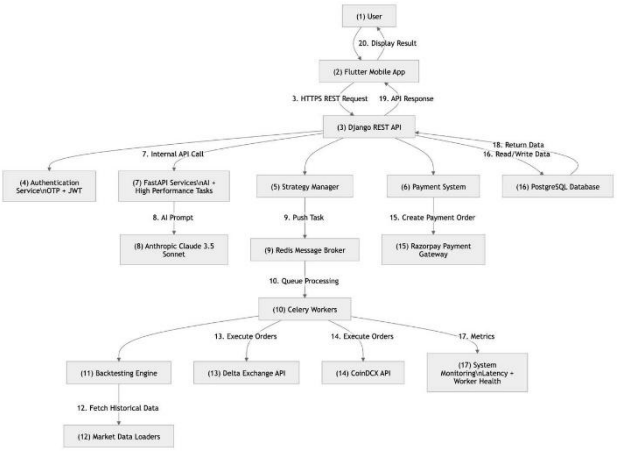

Fig. 1. System Architecture of the Proposed Platform

The proposed system is constructed on a distributed, service-oriented architecture [8] that partitions functionality into clearly bounded layers, each optimized for a specific category of operations. This separation of concerns enables each service to scale independently, simplifies maintenance, and allows individual components to evolve without introducing regressions in adjacent parts of the system [7].

User interaction is facilitated by a Flutter-based mobile application providing an intuitive interface for inputting trading strategies in natural language, reviewing AI-generated strategy outputs, and managing account and subscription details. All data exchanged between the client and backend travels exclusively through TLS-secured REST endpoints [9]. Core request orchestration is performed by a Django REST Framework service [9] that coordinates user authentication, strategy lifecycle management, payment processing [12], and routing between internal modules and third-party integrations. This layer enforces access control

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

policies and maintains a consistent data contract for all system consumers [8].

LLM inference is encapsulated within a standalone FastAPI microservice that communicates with the hosted language model [5] to process incoming natural language descriptions and emit structured strategy representations [4]. Isolating this service from the main backend enables independent scaling during inference-intensive periods and prevents AI-related latency from affecting other system functions. Computeintensive operations—primarily strategy simulation [1] and optimization workflows—are routed to a distributed processing tier built on Celery task workers and a Redis message broker [10]. Separating these workloads from the synchronous request path [7] preserves API responsiveness and allows the system to handle bursts of concurrent requests without visible queuing delays for end users.

Persistent state is managed by a PostgreSQL instance [11] that stores user profiles, strategy definitions, transaction histories, and operational logs in a normalized relational schema, supporting both transactional consistency and complex analytical queries. Real-time order submission and portfolio updates are handled through authenticated connections to the Delta Exchange [13] and CoinDCX [14] broker APIs. A supplementary monitoring layer continuously tracks service health—including endpoint latency, task queue depth, and worker availability [7]—providing the observability needed to detect and address performance anomalies before they impact users.

B. System Components

The following components collectively realize the capabilities of the proposed platform [8], each operating within a well-defined boundary and communicating with adjacent components through stable, versioned interfaces.

• Client Application: Built using the Flutter SDK, the mobile client is the primary interface for users. It presents an input surface for natural language strategy descriptions [4], displays AI-generated strategy summaries, shows simulation results [1], and supports account management. All client-server communication is secured through TLS-protected REST connections [9].

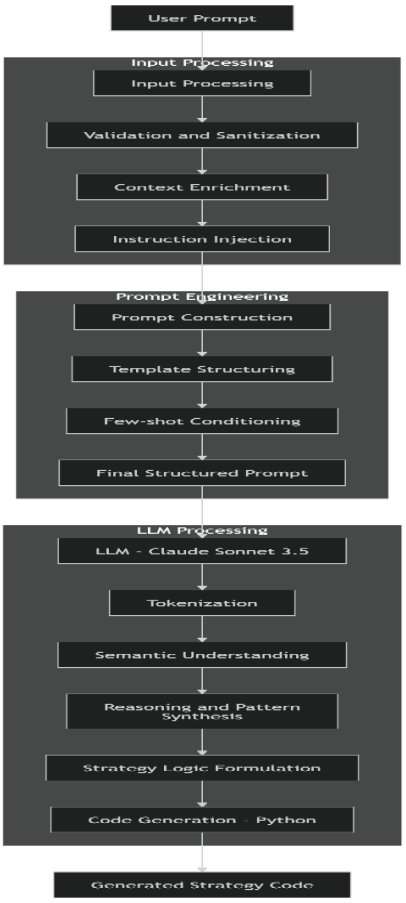

Fig. 2. LLM-Based Strategy Generation Pipeline

• Backend API Layer: The Django REST Framework [9] underpins the central orchestration service, which manages user identity, strategy persistence, request routing, payment handling [12], and third-party service integration [13], [14]. It acts as the authoritative coordinator across all subsystems [8], enforcing data integrity and role-based access control.

• AI Processing Component: A FastAPI service hosts the inference pipeline responsible for transforming free-text trading descriptions into structured strategy objects. It invokes the LLM [5] to extract trading primitives—including indicator configurations [1], entry and exit signal definitions, position sizing rules, and risk parameters [15]. Extracted fields are serialized into a canonical intermediate format (typically JSON) before being handed off to the validation module.

• Asynchronous Processing: A pool of Celery workers [10] consumes tasks from a Redis-backed queue and executes them concurrently in the background. This tier handles high-latency operations such as multi-period backtests [1] and

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

strategy parameter sweeps without occupying synchronous request threads [7], preserving system responsiveness under load.

• Data Storage: PostgreSQL [11] provides durable, ACIDcompliant persistence for all user-generated artifacts, including strategy definitions, execution logs, subscription records, and audit trails. The relational schema is optimized for both transactional write patterns and the analytical read queries required by reporting features.

• Backtesting Engine: This component replays validated strategies against historical price and volume data [1], [16], simulating order fills under configurable market assumptions. It computes a standardized suite of performance metrics—net profit/loss, peak-to-trough drawdown, and trade-level win rate [15]—providing a quantitative basis for evaluating strategy variants before committing real capital.

• Broker Integration: Authenticated REST connectors to Delta Exchange [13] and CoinDCX [14] enable the platform to place market and limit orders, query execution status, and synchronize portfolio state. The integration layer abstracts exchange-specific API differences behind a uniform internal interface, simplifying the process of adding new exchange connectors in the future.

• Payment and Credit Management: Razorpay [12] is integrated to process subscription purchases and credit topup transactions. An internal credit ledger tracks each user's available balance, which is decremented as premium features—such as advanced AI generation and extended historical simulations [1]—are consumed.

• Monitoring and Logging: Structured logs and performance telemetry are collected across all services, capturing API round-trip times, task durations, queue depths, and worker health indicators [7]. This observability layer supports proactive anomaly detection, root-cause analysis during incidents, and data-driven capacity planning.

C. Methodology

• Planning and Requirement Analysis: The project begins with a structured requirements elicitation phase that examines the shortcomings of existing algorithmic trading tools [8]. Functional requirements are derived from identified gaps: natural language strategy input [4], [5], automated strategy validation [2], historical performance simulation [1], asynchronous task scheduling [7], [10], and live broker connectivity [13], [14]. Non-functional requirements address throughput targets, fault tolerance, and extensibility.

• System Design: The architectural blueprint adopts a multitier, service-oriented model [8] in which cross-cutting concerns are resolved at layer boundaries. The frontend, core orchestration service [9], AI inference engine [5], and execution tier are treated as independently deployable units with explicit interface contracts. This approach ensures that improvements to any single component—particularly the LLM inference pipeline—can be delivered without disrupting dependent services [7].

• Implementation: Each tier is implemented using technolo- gies selected for their fit to the respective workload. Flutter enables consistent, responsive cross-platform UI rendering; Django REST [9] manages complex business logic and relational data access; FastAPI [5] provides low-overhead asynchronous handling suited to AI inference workloads; and the Celery-Redis combination [10] orchestrates background task execution. Each service is developed, unittested, and integration-tested independently before being incorporated into the composite system.

• AI Model Integration: Raw user input is forwarded to the LLM [5] through a structured prompt that instructs the model to extract a predefined set of trading primitives: indicator identifiers and parameters [1], directional signal conditions, order types, and risk control settings [15]. The model response is parsed into a validated schema object [4], which then proceeds through the validation pipeline [2] for constraint checking before any persistent storage or downstream processing is initiated.

• Testing and Evaluation: Verification proceeds through complementary testing regimes [7]. Unit tests confirm module-level correctness [2], while integration tests exercise cross-service communication paths. Load tests emulate concurrent user activity to characterize throughput and latency under realistic demand. The asynchronous

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

processing tier [10] receives particular attention during load testing to verify that task queue saturation does not degrade API response times during peak simulation workloads [1].

• Deployment: Production deployment targets a cloud-hosted environment with containerized service packaging. Docker images encapsulate each service's runtime dependencies, enabling reproducible deployments and straightforward horizontal scaling. Integrations with broker APIs [13], [14] and the payment gateway [12] are configured through environment-injected credentials. The iterative release cadence prioritizes the AI inference module [5], which benefits most from continuous refinement as real-world usage data accumulates, in alignment with established enterprise software practices [8].

D. PROJECT IMPLEMENTATION

The system implementation assembles a purposefully selected technology stack, with each component chosen based on its performance characteristics, maintainability, and compatibility within a distributed service architecture [8].

The user-facing mobile application is developed with Flutter and the Dart programming language. Flutter's single-codebase deployment model ensures visual and behavioral consistency across both iOS and Android without duplicating development effort. Its reactive widget system and efficient rendering engine are particularly well-suited to handling the real-time state updates that occur when strategy generation results and order status notifications arrive asynchronously from the backend [9].

The primary application backend is implemented using the Django REST Framework [9]. Django's mature ORM, integrated session management, and comprehensive middleware ecosystem simplify the implementation of authentication workflows, access control, and request serialization. The framework's well-defined project structure supports long-term code maintainability as the platform grows [8]. Payment reconciliation and subscription management are handled through an embedded Razorpay integration [12].

AI inference is delegated to a separate FastAPI microservice [5]. FastAPI's native support for asynchronous I/O via Python's asyncio runtime makes it well-suited to operations that spend the majority of their execution time awaiting responses from a remotely hosted language model. Isolating inference within its own service boundary prevents AI latency spikes from propagating to the synchronous API surface and allows this tier to scale independently when inference demand increases [7].

Relational data storage is provided by PostgreSQL [11], which offers strong transactional guarantees, rich indexing capabilities, and support for JSON column types that accommodate the semi-structured strategy definitions produced by the AI pipeline. Its proven stability under high-write production workloads makes it a sound foundation for audit-critical financial records.

Background task execution is coordinated by Celery [10] over a Redis message broker. Celery serializes task invocations as messages, which Redis queues and delivers to available worker processes running in parallel. This arrangement allows backtesting jobs [1]—which may involve processing millions of historical data points—to run concurrently without occupying request-serving processes, thereby preserving low API response times for interactive operations [7].

Exchange connectivity is implemented through authenticated HTTP clients targeting the Delta Exchange [13] and CoinDCX [14] REST and WebSocket APIs. Each connector implements the broker's authentication scheme, rate-limit backoff logic, and order status polling behind a common internal interface, allowing strategy execution to remain exchangeagnostic at the application layer.

Deployment packaging uses Docker to containerize each service, producing portable, environment-independent images that can be promoted uniformly from development through staging to production. This strategy simplifies dependency management, reduces environment-specific defects, and creates a clear path toward orchestrated multi-instance deployments [7]. The overall implementation consistently reflects the principles of modularity, measurable performance, and horizontal scalability [8].

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

IV. RESULTS AND DISCUSSION

System performance is assessed across four operational dimensions: strategy synthesis quality, validation effectiveness, simulation throughput, and asynchronous processing stability [8]. Each dimension is evaluated through targeted experiments designed to isolate the contribution of the relevant subsystem [7].

• AI Strategy Generation: Strategy synthesis latency and structural correctness are the primary evaluation criteria for the LLM inference module [5]. Across a representative test corpus spanning a range of input complexities, the system produced well-formed strategy objects within an average response window of 1–2 seconds per request. Review of the generated outputs confirmed accurate extraction of indicator specifications [1], directional entry and exit conditions, and risk parameters [15] in the overwhelming majority of evaluated cases, validating the practical viability of natural language as a strategy authoring medium.

• Strategy Validation: The constraint-checking module [2] was evaluated on its capacity to identify structural and logical defects in AI-generated strategies before they progress to simulation or execution. Over the evaluation suite, the validation engine correctly flagged missing signal conditions, internally contradictory rule combinations, and out-ofbounds parameter values in approximately 92% of injected test cases. This detection rate meaningfully reduces the risk of deploying strategies containing latent errors that could generate unintended market positions [15].

• Backtesting Engine: The simulation component [1], [16] was benchmarked using historical price series across multiple instruments and timeframes. End-to-end simulation of a single strategy completed within 3–6 seconds, with execution time scaling predictably as dataset length and strategy complexity increased. Performance metrics—including net profit/loss, maximum drawdown, and trade-level win rate [15]—were computed accurately across all tested configurations and rendered as structured reports accessible from the client interface. Concurrent simulation requests submitted during load tests were handled without measurable degradation in per-job throughput, confirming the effectiveness of the asynchronous dispatch architecture [10].

Fig. 3. Backtesting Chart with Buy/Sell Signals

• Asynchronous Processing: The Celery-Redis task execution tier [10] was assessed under sustained concurrent load representative of expected production traffic. Across all test scenarios, the system maintained stable task throughput and negligible queue accumulation, with no measurable increase in API response latency attributable to background task activity [7]. Isolating compute-intensive jobs within dedicated worker processes effectively shields the interactive request path from resource contention.

• Execution: The broker integration layer [13], [14] was validated in controlled sandbox environments provided by each exchange. Order submission, acknowledgement, and status tracking functioned reliably within latency bounds acceptable for near-real-time cryptocurrency trading [15]. The error-handling subsystem correctly classified and recovered from transient API failures and network interruptions, preventing partial state inconsistencies from propagating to strategy execution records.

Metric	Value
Net Profit/Loss	$-10163.92
Win Rate	32.29%
Total Trades	96
Max Drawdown	100%
Sharpe Ratio	-0.24
Profit Factor	0.53

TABLE I
PERFORMANCE METRICS OF GENERATED STRATEGY

As shown in Table I, the generated trading strategy produced a net loss, a low win rate, and high

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

drawdown. These results indicate that while the system successfully translates natural language into executable strategies, further optimization and more robust risk management mechanisms are needed to improve financial outcomes.

Taken together, the experimental findings confirm that the system successfully meets its core design objective: making strategy creation accessible to non-technical users [4], [5] while sustaining the performance and reliability characteristics required of a production trading platform [7], [8]. A notable caveat is that execution latency remains partially dependent on third-party broker API stability [13], [14] and network conditions—an irreducible source of variability beyond the system's direct control. Notwithstanding this constraint, the evaluation outcomes provide strong evidence that the proposed platform constitutes a practical and accessible solution for its target user base [16].

V. CONCLUSION

This paper has introduced an AI-powered algorithmic trading platform that applies Large Language Models (LLMs) to enable users to specify, validate, simulate, and deploy trading strategies through natural language descriptions [4], [8]. The work is motivated by a well-recognized gap between the trading intuition of retail investors and the technical prerequisites imposed by existing algorithmic tools [1], [16].

Central to the system is an LLM [5] that autonomously converts informal user intent into formally structured trading logic, removing the requirement for programming literacy and substantially widening the pool of users who can participate in algorithmic trading [15], [16]. A dedicated constraint-checking layer [2] enforces logical completeness and risk compliance on every generated strategy, serving as a critical quality gate before simulation or live deployment proceeds. The asynchronous task architecture powered by Celery and Redis [10] absorbs peak computational demand without compromising interface responsiveness, while horizontal scalability [7] ensures the platform can accommodate a growing user population.

Integrated backtesting [1] equips traders with evidence-based performance feedback prior to capital commitment, and direct broker connectivity [13], [14] closes the loop from strategy conceptualization to live market execution. Supplementary modules for payment handling [12] and operational monitoring [7] complete a platform designed for both robustness and usability.

Quantitative evaluation confirms that the system meaningfully reduces strategy development friction [8] while preserving the performance and reliability standards required in financial production environments [7]. By combining AI-driven generation [5] with a principled, scalable engineering foundation [8], the proposed platform charts a practical course toward inclusive, intelligent algorithmic trading [15], [16].

In summary, this work establishes that combining modern language models with distributed software architecture can substantially lower the threshold for algorithmic market participation [16] without compromising system integrity. The modular design [8] ensures the platform remains adaptable as advances in AI [3], [6] and financial technology [15] continue to reshape the trading landscape.

Several directions for future development can enhance the platform's adaptability. Reinforcement learning techniques may be incorporated to enable dynamic strategy optimization in response to changing market conditions [3], [6]. The natural language processing module can be further improved through domain-specific fine-tuning of LLMs [4], [5] to increase the accuracy and consistency of strategy generation. Additionally, the backtesting engine can be extended to incorporate more realistic market conditions, including transaction costs, market impact, and slippage [15]. Expanding asset class support and integrating additional broker APIs [13], [14] would further increase the system's reach and applicability across trading contexts.

ACKNOWLEDGMENT

The authors express sincere gratitude to all individuals and institutions whose guidance, feedback, and infrastructure support contributed to the successful completion of this research.

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.

REFERENCES

[1] J. Welles Wilder, "New Concepts in Technical Trading Systems," Trend Research, 1978.

[2] T. M. Mitchell, "Machine Learning," McGraw-Hill, 1997.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," Nature, vol. 521, no. 7553, pp. 436–444, 2015.

[4] A. Vaswani et al., "Attention Is All You Need," Advances in Neural Information Processing Systems (NeurIPS), 2017.

[5] OpenAI, "GPT-4 Technical Report," 2023.

[6] F. Chollet, "Deep Learning with Python," Manning Publications, 2017.

[7] M. L. Pinedo, "Scheduling: Theory, Algorithms, and Systems," Springer, 2016.

[8] M. Fowler, "Patterns of Enterprise Application Architecture," Addison-Wesley, 2002.

[9] Django Software Foundation, "Django Documentation," https://docs.djangoproject.com/

[10] Celery Project, "Celery Distributed Task Queue Documentation," https://docs.celeryq.dev/

[11] PostgreSQL Global Development Group, "PostgreSQL Documentation," https://www.postgresql.org/docs/

[12] Razorpay, "Payment Gateway Documentation," https://razorpay.com/docs/

[13] Delta Exchange, "API Documentation," https://www.delta.exchange/

[14] CoinDCX, "API Documentation," https://coindcx.com/

[15] J. Hull, "Options, Futures, and Other Derivatives," Pearson, 2018.

[16] E. F. Fama, "Efficient Capital Markets: A Review of Theory and Empirical Work," Journal of Finance, 1970.

© Author(s). This work is peer-reviewed, openly published, and permanently archived
This article is openly accessible and reusable with proper attribution.
https://ijsmt.org/ , Email: editor@ijsmt.org