Citadel – Build The Algorithmic Black Box

Kanishk Choudhary

February 1, 2026

Contents

Chapter 1

Introduction

1.1 Project Objectives

Modern financial markets are dominated by algorithmic trading systems whose internal logic is often opaque. This project, titled *Citadel – Build The Algorithmic Black Box*, aims to demystify such systems by constructing a complete simulated electronic market and embedding an intelligent trading agent within it.

The primary objectives are:

- To design a realistic limit order book from first principles
- To simulate heterogeneous market participants
- To train a reinforcement learning agent capable of interacting with the market
- To analyze learned trading behavior under different market conditions

1.2 Report Organization

This report is structured as follows. Chapter 2 reviews relevant literature. Chapter 3 introduces the theoretical foundations. Chapter 4 describes the system architecture. Chapter 5 presents the agent ecology. Chapter 6 details the reinforcement learning implementation. Chapter 7 discusses experiments and results. Chapter 8 concludes with insights and future work.

Chapter 2

Literature Review

2.1 Agent-Based Models in Finance

Agent-based models (ABMs) have long been used to study financial markets as complex adaptive systems. Unlike equilibrium-based models, ABMs allow heterogeneous agents to interact locally, producing emergent global phenomena such as volatility clustering and fat-tailed return distributions.

2.2 Limit Order Books (LOB)

The continuous double auction limit order book is the dominant market mechanism in modern exchanges. Extensive research has examined its dynamics, including order flow imbalance, spread formation, and liquidity provision. Simulating an LOB is essential for studying realistic trading behavior.

2.3 Reinforcement Learning for Trading

Recent advances in reinforcement learning have enabled agents to learn trading strategies directly from interaction. However, many studies rely on historical price series and ignore microstructure. This project focuses on learning *within* the market.

Chapter 3

Theoretical Framework

3.1 Stochastic Modeling of Asset Prices

3.1.1 The Random Walk Hypothesis

Financial asset prices are commonly modeled as stochastic processes. The random walk hypothesis states that successive price changes are independent and identically distributed, implying that past information cannot be used to predict future returns.

Let P_t denote the asset price at time t. The log-return is defined as:

$$r_t = \log(P_t) - \log(P_{t-1})$$

In a pure random walk, r_t is assumed to be independent with zero mean.

3.1.2 Geometric Brownian Motion (GBM)

A widely used continuous-time model is Geometric Brownian Motion:

$$dP_t = \mu P_t dt + \sigma P_t dW_t$$

where μ is the drift, σ is volatility, and W_t is a Wiener process.

Although GBM is not directly simulated in the limit order book, it motivates the fair-value process used by agents.

3.1.3 Ito's Lemma

Ito's Lemma allows analytical treatment of functions of stochastic processes. For a function $f(P_t, t)$:

$$df = \frac{\partial f}{\partial t} dt + \frac{\partial f}{\partial P} dP_t + \frac{1}{2}\frac{\partial^2 f}{\partial P^2}(dP_t)^2$$

—

3.2 Market Microstructure Theory

3.2.1 The Continuous Double Auction

Modern exchanges operate via a continuous double auction (CDA), where buy and sell orders arrive asynchronously and are matched continuously.

At any time t:

$$\text{Spread}_t = \text{BestAsk}_t - \text{BestBid}_t$$

The mid-price is defined as:

$$\text{Mid}_t = \frac{\text{BestBid}_t + \text{BestAsk}_t}{2}$$

3.2.2 Liquidity and Market Depth

Liquidity is reflected by:

- Tight bid–ask spreads
- High depth near the mid-price

Market makers supply liquidity by continuously quoting both sides of the book.

Chapter 4

System Architecture

4.1 The Matching Engine

4.1.1 Order Book Data Structures

The order book is implemented using priority queues for bids and asks, enforcing price–time priority.

4.1.2 Latency Simulation

Order submission latency is modeled using an exponential distribution, introducing realistic asynchrony.

4.2 Software Implementation Details

The system is written in Python and structured into modular components: engine, environment, agents, and analytics.

4.3 Core Functions of the Matching Engine

This section explains key functions used in the implementation of the market simulator. Each function is designed to model a specific real-world exchange mechanism.

4.3.1 `submit(order)`

The `submit` function is the primary entry point for all orders entering the market. It performs three sequential operations:

1. Attempts to match the incoming order against existing liquidity

2. Adds any remaining quantity to the order book
3. Records a snapshot of the updated book state

Formally, let q_{in} denote the incoming order quantity. If matching liquidity exists, trades are generated until either $q_{in} = 0$ or price constraints are violated.

This function ensures that market orders are executed immediately, while limit orders may rest in the book.

4.3.2 `_match(incoming)`

The `_match` function implements the price–time priority matching logic. For a buy order, it repeatedly compares the incoming price p_b against the best ask p_a:

$$\text{Match occurs if } p_b \geq p_a$$

For sell orders, the symmetric condition applies. The traded quantity is computed as:

$$q = \min(q_{\text{incoming}}, q_{\text{book}})$$

Partial fills are supported, and unmatched quantities are either reinserted into the heap or discarded in the case of market orders.

4.3.3 `current_snapshot()`

This function aggregates raw heap entries into price levels, producing a clean representation of the order book. Internally, prices are grouped using hash maps before being sorted.

The snapshot exposes:

- Best bid and best ask
- Aggregated depth at each price level

This abstraction decouples low-level heap storage from higher-level analytics.

The system is written in Python and structured into modular components: engine, environment, agents, and analytics.

4.4 Computational Complexity Analysis

This section analyzes the computational complexity of the core components of the trading system. Understanding time and space complexity is essential for evaluating scalability, performance bottlenecks, and realism of the simulated market.

4.4.1 Order Book Operations

The limit order book is implemented using two priority queues:

- A max-heap for buy orders (bids)
- A min-heap for sell orders (asks)

Let N denote the total number of active orders in the book.

Order Insertion

Inserting a new limit order into a heap requires maintaining heap order. The time complexity is:

$$O(\log N)$$

This operation is efficient and consistent with real-world exchange implementations.

Best Price Access

Accessing the best bid or best ask corresponds to inspecting the root of the heap:

$$O(1)$$

This enables constant-time retrieval of top-of-book prices, which is critical for fast matching and agent decision-making.

Order Matching

Matching an incoming order against the opposite side of the book may require multiple heap removals. In the worst case, an incoming order matches against all resting orders:

$$O(N \log N)$$

However, in typical market conditions, matching depth is shallow, and average-case complexity is significantly lower.

Order Cancellation

Order cancellation requires scanning the heap to locate the target order:

$$O(N)$$

After removal, heap rebalancing costs $O(\log N)$. While cancellation is relatively expensive, it occurs infrequently compared to submissions and matches.

—

4.4.2 Snapshot Generation

Order book snapshots aggregate orders by price level using hash maps.

Aggregation by Price

Each order in the heap is processed once:

$$O(N)$$

Hash-based aggregation ensures constant-time insertion per order.

Sorting Price Levels

After aggregation, price levels are sorted:

$$O(K \log K)$$

where K is the number of distinct price levels, typically $K \ll N$.

—

4.4.3 Event-Driven Simulation Engine

The simulation engine schedules all market events using a global priority queue ordered by event time.

Event Scheduling

Each event insertion into the event queue requires:

$$O(\log M)$$

where M is the number of scheduled events.

Event Execution

Removing the next event from the queue is:

$$O(\log M)$$

This design ensures chronological execution and deterministic behavior.

—

4.4.4 Agent Decision Logic

Noise Traders

Noise trader decision logic consists of random sampling and simple arithmetic:

$$O(1)$$

Market Makers

Market makers compute bid and ask quotes using current mid-price and inventory:

$$O(1)$$

However, cancel-replace behavior may generate multiple order cancellations and insertions, increasing operational load on the order book.

Momentum Traders

Momentum traders maintain a rolling price window of size W using a deque. Updating the moving average costs:

$$O(1)$$

due to bounded window size.

—

4.4.5 Reinforcement Learning Environment

Observation Construction

Building the state vector involves extracting the top D levels from both sides of the book:

$$O(D)$$

Since D is fixed, this operation is effectively constant time.

Reward Computation

Reward calculation involves arithmetic operations on portfolio value and drawdown:

$$O(1)$$

Environment Step

Each environment step includes action execution, reward computation, and observation update:

$$O(\log N)$$

dominated by order submission and matching.

—

4.4.6 Overall System Complexity

Let T denote the total number of simulation steps. The overall complexity of a full run is approximately:

$$O(T \log N)$$

This makes the system scalable to long simulations while preserving realistic microstructure behavior.

4.4.7 Space Complexity

The dominant memory components are:

- Order book heaps: $O(N)$
- Event queue: $O(M)$
- Logger data structures: $O(T)$

Total space complexity grows linearly with simulation length and market activity.

—

4.4.8 Discussion

The use of heap-based priority queues and event-driven simulation ensures that the system remains computationally efficient while retaining key properties of real financial markets.

Although some operations, such as order cancellation, are costly, they reflect realistic exchange constraints and trade-offs.

This complexity profile validates the architectural design and supports future extensions to higher agent counts and longer simulation horizons.

Chapter 5

Agent Ecology

5.1 Noise Traders

Noise traders submit random market and limit orders, injecting liquidity and randomness.

5.2 Market Makers

Market makers quote both sides of the book and profit from the bid–ask spread.

5.2.1 Inventory Skew Logic

Quotes are adjusted based on inventory to reduce risk exposure.

5.3 Momentum Traders

Momentum traders follow short-term trends, generating directional pressure.

5.4 Agent Decision Functions

Each agent class implements a `get_action` function that maps the current market state to an action. This design mirrors decision-making in real trading systems.

5.4.1 Noise Trader `get_action`

Noise traders generate stochastic order flow. Given the current fair value F_t, a noise trader randomly selects:

- Trade direction (buy or sell)

- Order type (market or aggressive limit)
- Quantity within a fixed bound

This behavior injects liquidity and randomness into the market, preventing degenerate dynamics.

5.4.2 Market Maker `get_action`

The market maker computes bid and ask quotes around the mid-price:

$$\text{Bid} = \text{Mid} - \frac{s}{2} - \alpha I_t$$

$$\text{Ask} = \text{Mid} + \frac{s}{2} + \alpha I_t$$

where s is the base spread, α is the inventory skew coefficient, and I_t is current inventory.

This logic enforces inventory mean reversion by discouraging accumulation on one side.

5.4.3 Momentum Trader `get_action`

Momentum traders maintain a rolling window of past prices and compute a simple moving average:

$$\text{SMA}_t = \frac{1}{N} \sum_{i=1}^{N} P_{t-i}$$

If the current mid-price exceeds the SMA, the trader buys; otherwise, it sells. This introduces directional pressure into the market.

Chapter 6

Reinforcement Learning Implementation

6.1 Markov Decision Process

The trading problem is modeled as a Markov Decision Process (MDP) defined by (S, A, R, P).

6.1.1 State Space (S)

The state vector includes:

- Normalized bid and ask prices
- Corresponding order sizes
- Inventory level I_t
- Cash balance C_t

6.1.2 Action Space (A)

$$A = \{\text{Hold}, \text{Buy}, \text{Sell}\}$$

6.1.3 Reward Function (R)

Portfolio value:

$$V_t = C_t + I_t \cdot \text{Mid}_t$$

Change in value:

$$\Delta V_t = V_t - V_{t-1}$$

Drawdown:

$$D_t = \max_{s \le t} V_s - V_t$$

Final reward:

$$R_t = \Delta V_t - c \cdot \mathbb{I}(\text{trade}) - \lambda D_t$$

This formulation penalizes excessive risk and overtrading.

6.2 Proximal Policy Optimization

PPO maximizes the clipped objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

This prevents destructive policy updates.

6.3 Trading Environment Functions

The reinforcement learning environment encapsulates market interaction logic and ensures compatibility with Gymnasium.

6.3.1 `reset()`

The `reset` function initializes a new episode by:

- Clearing the order book
- Resetting inventory and cash
- Reinitializing the simulation engine

This guarantees independent episodes and reproducibility under fixed seeds.

6.3.2 `step(action)`

The `step` function applies the agent's action to the market. If the agent selects a buy or sell action, a market order of unit quantity is submitted.

After execution, the environment computes:

- Updated inventory I_t
- Updated cash C_t

- Portfolio value V_t
- Reward R_t

Termination occurs if inventory or cash constraints are violated.

6.3.3 `calculate_reward()`

Reward computation is separated for clarity. The function computes:

$$V_t = C_t + I_t \cdot \text{Mid}_t$$

and penalizes drawdowns:

$$D_t = \max(V_s) - V_t$$

This modular structure simplifies experimentation with alternative reward designs.

Chapter 7

Experiments and Results

7.1 Verification of Market Dynamics

7.1.1 Volatility Clustering

Returns are computed as:

$$r_t = \log(\mathrm{Mid}_t) - \log(\mathrm{Mid}_{t-1})$$

Periods of elevated variance appear in clusters, consistent with empirical markets.

7.1.2 Heavy-Tailed Returns

Return distributions exhibit excess kurtosis:

$$\kappa = \frac{E[(r_t - \mu)^4]}{\sigma^4} > 3$$

7.2 RL Agent Performance

7.2.1 Equity Curve

The equity curve V_t stabilizes over training, indicating convergence.

7.2.2 Inventory Control

Inventory remains bounded:

$$|I_t| \leq I_{\max}$$

This demonstrates implicit risk management.

7.3 Validation and Sanity Check Functions

To ensure correctness, multiple validation routines are used.

7.3.1 `validate_book_snapshot`

This function verifies that:

- Prices and quantities are non-negative
- Best bid is strictly less than best ask

Violations indicate logical errors in matching or order handling.

7.3.2 `validate_pipeline`

This function performs end-to-end checks, including:

- Volume-weighted average price (VWAP) bounds
- Non-negative spread
- Reasonable volatility relationships

These checks confirm that emergent market statistics remain plausible.

Chapter 8

Conclusion and Future Work

8.1 Summary

This project demonstrates that reinforcement learning agents can learn meaningful trading behavior in a simulated limit order book.

8.2 Future Directions

Future extensions include richer action spaces, transaction fees, and multi-agent learning.

Appendix A

Appendix: Code Listings

A.1 The Matching Engine (Python)

A.2 Agent Logic: Market Maker

A.3 PPO Reward Function