Financial advisor Bot

Final project

Total word Count: 9444

Table of contents

1 Intro

Word Count: 503

I selected the ***Financial Advisor Bot*** project template in order to apply and critically evaluate Artificial Intelligence techniques studied during the degree within a complex and realistic real-world domain: financial markets.

I chose this project because I wanted to move beyond controlled academic environments and explore how learning-based agents behave in settings that are noisy, non-stationary, and partially unpredictable. Financial markets present a particularly challenging problem for reinforcement learning: decision outcomes are delayed, reward signals are ambiguous, and strategies that perform well in one market regime may quickly become ineffective in another. These characteristics make financial decision-making an appropriate and demanding test for adaptive AI systems.

During the Artificial Intelligence course, I studied core learning mechanisms such as backpropagation, loss optimisation, reinforcement learning, and genetic algorithms. Among these, I found Reinforcement Learning (RL) and Genetic Algorithms (GA) very interesting and relevant for problems that require continual adaptation through interaction and feedback. Practical coursework, including the implementation of Deep Q-Network agents using OpenAI Gym environments, demonstrated how agents can learn effective policies through trial and error in simulated tasks such as Atari games or control problems.

The Financial Advisor Bot project is based on reinforcement learning, genetic optimisation, and explainable decision-making: it builds directly on those foundations but applies the same principles to a far more complex domain. Unlike games or simulated environments with fixed rules and clear success criteria, financial markets are uncertain, noisy, and hence very challenging. This contrast highlights a central question that motivates the project: whether an intelligent agent can extract meaningful signals from seemingly chaotic data through iterative learning rather than explicit rule design.

One analogy related to Artificial Intelligence that I find particularly relevant is the idea that

“Machine Learning is like giving an algorithm a crumpled piece of paper and asking it to un-crumple it.”

In the context of this project, financial markets represent that crumpled structure: a system where useful patterns may exist but are obscured by noise, volatility, and randomness. The Financial Advisor Bot is therefore conceived as an attempt to investigate how learning-based agents can progressively uncover structure in such data through experience.

In addition to reinforcement learning, this project explores the use of Genetic Algorithms as a complementary optimisation mechanism to support adaptive decision-making. Genetic optimisation is considered as a way to tune parameters and decision thresholds in a domain where manual calibration is difficult and challenging.
The project also aims to incorporate an explainability component through a Natural Language Processing layer, motivated by the importance of transparency and interpretability in financial decision-support systems. Users must be able to understand not only *what* recommendation is produced, but *why* it is suggested.

Overall, this project is designed to apply, extend, and critically assess Artificial Intelligence concepts studied during the course within a demanding real-world setting. By combining reinforcement learning, evolutionary optimisation, and explainable outputs, the Financial Advisor Bot serves both as a technical challenge and as a structured investigation into the strengths and limitations of modern AI methods when applied beyond controlled academic environments.

2 Literature review

Word Count: 2471

For the literature review, I studied key research applying Reinforcement Learning (RL) and Genetic Algorithms (GA) to financial trading and portfolio optimisation. These two techniques were central to my studies during the degree, and they form the theoretical foundation of my project. By analysing how prior work has modelled trading as a sequential decision-making problem, designed reward functions, addressed market volatility, and integrated evolutionary optimisation, the aim for the review is to identify both effective strategies and related strengths and weaknesses. The selected studies range from early direct-reinforcement methods to recent deep RL, hybrid GA-RL, and multi-module architectures.

2.1 Moody & Saffell (2001)

The first example I encountered during my background and literature research dates back to the early 2000s, when **Moody and Saffell (2001)** presented one of the earliest RL approaches for trading, introducing a *direct reinforcement (DR)* framework in which an agent optimises portfolio performance without relying on value-function estimation. Their method formulates trading as a stochastic control problem and updates policy parameters through gradient ascent on a utility-based objective.

A key contribution of their work is the explicit avoidance of the recursive Bellman equation to circumvent the “curse of dimensionality.” Instead of predicting long-term value estimates as in Q-learning, their system optimises a risk-adjusted performance

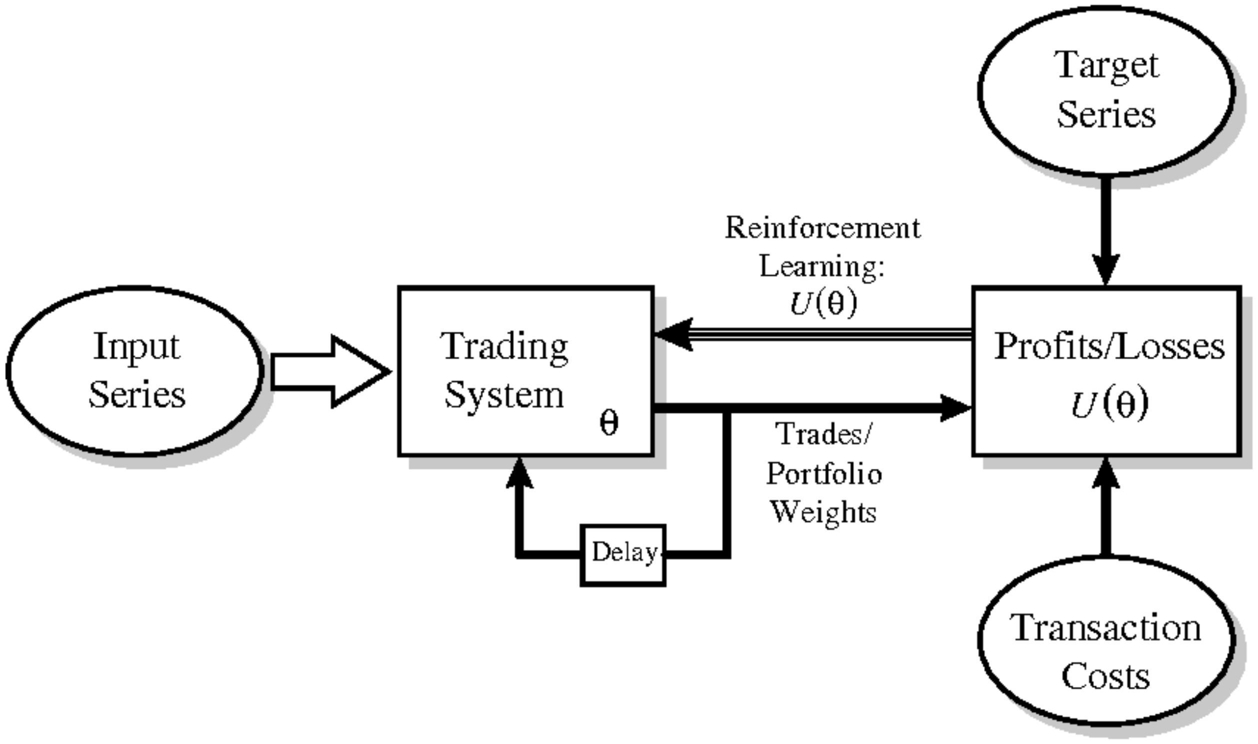

Target Series
Reinforcement Learning: $U(\theta)$
Input Series
Trading System
θ
Profits/Losses $U(\theta)$
Trades/ Portfolio Weights
Delay
Transaction Costs

Figure 1: RL overview

measure directly (the differential Sharpe ratio). This insight reflects an essential principle in financial RL: reward functions must incorporate volatility, risk, and transaction costs to remain viable beyond theoretical settings.

The paper also highlights early recognition of domain-specific constraints in trading systems, including the need for smooth, stable policy updates and the importance of reward shaping. Although computationally modest by modern standards, Moody and Saffell established conceptual foundations for later policy-gradient and model-free approaches in finance.

2.2 Deep Reinforcement Learning and Portfolio Optimisation - Jiang, Xu and Liang (2017)

A major development came with Jiang, Xu and Liang (2017), who introduced the **Ensemble of Identical Independent Evaluators (EIIE)** architecture for multi-asset portfolio allocation. Their system uses a collection of small neural networks (one per asset) that process historical price data and jointly output portfolio-weight adjustments. This structure is modular and linearly scalable: the model can incorporate new assets simply by adding additional evaluators.

The reward function is defined as portfolio growth at each time-step, enabling faster training and a mathematically clean formulation. However, the authors acknowledge that their assumptions, such as perfect liquidity, absence of slippage, and deterministic price execution, make the reported profitability over-optimistic compared to real markets.

I find it useful to compare these approaches with the ones I studied during my degree. In this case, I observed that the **Atari DQN agent** I implemented uses **one network** to predict action values for a single agent in a single environment, while **Jiang et al.'s model** uses **multiple identical networks** operating in parallel (each focused on one asset) and a shared top layer that merges their evaluations into portfolio actions.
In simple terms, DQN learns "what move to make next," while EIIE learns "how much to invest in each asset."

When comparing this solution not only with DQN but also with **Moody & Saffell (2001)**, I also noticed important differences in how **rewards** are defined:

- **Moody & Saffell** optimised a *risk-adjusted utility function* (the differential Sharpe ratio) , effectively teaching the agent to balance return and volatility.
- **Atari DQN** used a *delayed reward* system because you only know if the strategy works after multiple steps (e.g., finishing a level).
- **Jiang et al.**, by contrast, used an explicitly defined and immediate reward function based on portfolio growth at each step.

Their framework outperformed several traditional portfolio strategies, however Jiang et al. also acknowledged important **limitations**:

- Their system assumes no transaction delays or market impact, which is unrealistic for live trading.
- It does not handle long-term dependencies beyond a few timesteps.
- The model's success depends heavily on the stability of historical correlations, which is a major issue in volatile markets.

Reading this paper helped me appreciate how far RL has evolved since the early *direct reinforcement* methods. **Jiang et al.** extended the concept into a deep-learning framework capable of handling multiple assets simultaneously, combining scalability, modularity, and the ability to incorporate domain-specific constraints such as transaction costs.
To me, this work also reinforced one of the biggest challenges in applying deep RL to finance: **overfitting**. Financial markets are inherently **non-stationary**, meaning that yesterday's optimal policy can quickly become obsolete.

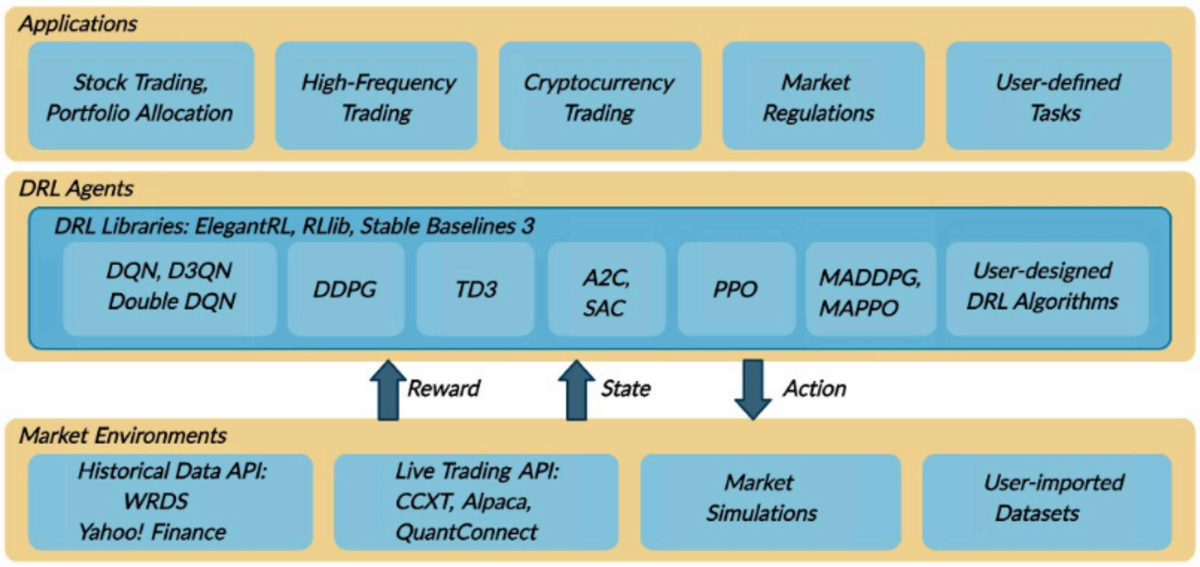


Figure 2: RL agent interacting with market env

2.3 FinRL – A Standardised Deep Reinforcement-Learning Framework for Financial Trading

After reviewing early Reinforcement Learning approaches, I wanted to explore what current researchers actually use to build trading agents. This search led me to **FinRL (Liu et al., 2021)**, an open-source library that provides ready-to-use environments, algorithms, and evaluation tools for financial reinforcement learning (FinRL is basically what **OpenAI Gym** is for games: a research platform that standardises environments, datasets, and reward functions so that different RL models can be compared under consistent scenarios).

One of FinRL's main goals is not to invent a new trading algorithm but to **unify the workflow** of reinforcement learning in finance. It wraps data sources such as *Yahoo Finance* and *Alpaca*, preprocesses price and fundamental data into market "states" and allows users to train standard RL agents, such as DQN, under consistent settings.

The environment simulates a trading session in which an agent observes features (e.g., prices, holdings, and cash balance) and chooses actions (buy, sell, or hold). After each step, the environment returns a reward (typically the change in portfolio value).
I immediately noticed the familiarity with the **Atari DQN** structure I studied during the course: the same action loop, replay buffer, and episodic learning cycle based on state observation (rewards are no longer game scores but **portfolio returns**, often adjusted for risk using techniques, like the **Sharpe ratio).**

Overall, I found FinRL to be an **excellent teaching and research tool.** Its modular design and clear documentation made it easy for me to relate to the frameworks I used during my coursework. The library integrates multiple reinforcement-learning paradigms (e.g. value-based, policy, etc.) and directly connects RL with financial evaluation metrics.

Despite its strengths, FinRL also illustrates again the main limitations of most current RL research in finance: although FinRL can simulate transaction costs, liquidity, order execution, and slippage are often ignored.

When comparing with Moody & Saffell (2001), I think FinRL generalises the concept of reinforcement learning from a single-agent system to a **broader framework** supporting many algorithms (some value-based like DQN, others policy-gradient based) whilst Moody and Saffell trained a single recurrent model to optimise a risk-adjusted objective.
In contrast to Jiang et al. (2017), FinRL does not propose a new neural topology such as EIIE: it implements existing ones within a controlled environment and it lacks Jiang et al.'s focus on multi-asset scalability and explicit transaction-cost handling.

Finally, compared with the **Atari DQN** I studied, FinRL formalises the same reinforcement-learning principles but applies them to a far noisier and less predictable environment. The agent still seeks to maximise cumulative reward through exploration and exploitation, but here the reward is tied to *real-world financial performance* rather than a deterministic score. This makes training more unstable and heavily dependent on **hyperparameter tuning** - an area where **Genetic Algorithms (GA)** could offer valuable support by automatically evolving parameters or reward functions, which is one of the directions I intend to explore further.

2.4 Hybrid Genetic-Algorithm and Deep Q-Learning Approaches (Ding et al., 2019; Zeng et al., 2023)

Given the volatility and non-stationarity of financial markets, several researchers have explored combining RL with Genetic Algorithms to enhance adaptability. Two notable examples are Ding et al. (2019) and Zeng et al. (2023), who introduce hybrid frameworks that evolve RL parameters using evolutionary search.

Ding et al. (2019) treat each DQN agent as an individual within a GA population. After each episode, agents are evaluated on cumulative and risk-adjusted returns, and the top performers undergo crossover and mutation to produce new policies. Over generations, this evolutionary process refines hyperparameters and stabilises the learning behaviour.

Zeng et al. (2023) extend this idea by evolving hyperparameters such as learning rate, exploration ratio, and discount factor dynamically throughout training. Their method reduces manual tuning and improves convergence in noisy environments.
These hybrid designs are conceptually compelling for several reasons:

- RL learns within a generation through experience replay and policy updates.
- GA evolves across generations by exploring broader design spaces.
- The combination supports both exploitation (learning from data) and exploration (searching for new strategies).

The main drawbacks are computational cost (training populations of DQN agents is resource-intensive) and reduced explainability. Nevertheless, these studies illustrate a promising direction for building more adaptive and robust financial decision systems, addressing limitations observed in purely RL-based frameworks such as FinRL.

2.5 ProTrader-RL (2024): Multi-Module Reinforcement Learning for Stock Trading

One of the most recent and relevant studies identified during my research is *ProTrader-RL* (2024), a framework designed to emulate the behaviour and reasoning process of professional traders by dividing the decision-making task into specialised modules.
The system is composed of **four main components**:

1. **Data Preprocessing**
2. **Buy Knowledge RL**
3. **Sell Knowledge RL**
4. **Stop-Loss Rule**

Each of these plays a distinct role that mirrors the workflow of human traders.
In their experiments, the authors reported strong and stable results, consistently high returns and Sharpe ratios with low maximum drawdown (MDD) across different market conditions outperforming prior reinforcement-learning-based trading systems.

Unlike earlier frameworks such as Jiang et al.'s (2017) EIIE model, which focused on allocating assets based on historical prices, **ProTrader-RL does not restrict its state inputs to specific stocks or fixed financial indicators**. Instead, it processes a broader spectrum of market data, representing the approach that professional traders typically adopt. The system primarily relies on **technical indicators and price series**, assuming that these features dominate short-term trading behaviour. However, the modular design leaves room to incorporate other signals, such as sentiment or macroeconomic factors.

Technically, ProTrader-RL is implemented within the **OpenAI Gym** environment and the **policy network** follows a **Deep Neural Network (DNN)** architecture with three hidden layers (69–40–2 units), used identically for both actor and critic networks. As reported in the '*Pro Trader RL: Reinforcement learning framework for generating trading knowledge by mimicking the decision-making patterns of professional traders*' article (it's mentioned in the reference) a total of sixty-nine variables are initially normalised and passed into the **two reinforcement-learning environments**. The reward function can be configured to maximise not only returns but also **risk-adjusted metrics** such as the Sharpe or Sortino ratios, allowing the agent to learn more stable and consistent trading strategies.

Compared with the approaches I studied earlier, I do believe that ProTrader-RL introduces several conceptual improvements. For example Moody & Saffell (2001) handled trading as a single-agent optimisation problem, while Jiang et al. (2017) extended this to multi-asset allocation: ProTrader-RL explicitly "splits" trading into multiple agents (aka "knowledge modules"), each optimising a different decision layer. This architecture not only reduces the learning complexity of each component but also mirrors human cognitive division of tasks (buying, selling, and risk control).
Also, from a systems perspective, this multi-module structure offers scalability and interpretability, characteristics that are often missing in several RL models such as FinRL's standard environments.

For my own project, this study is particularly inspiring: I intend to extend the **multi-agent collaboration concept / architecture** by integrating **parallel multiprocessing** (I will discuss this in more detail in the next section) and adding a **Sentiment Analysis agent** to validate Buy signals based on real-world news tone.

Furthermore, I plan to use **Genetic Algorithms** to evolve the hyperparameters of each RL module automatically, something not yet addressed in ProTrader-RL, which could make the overall system more adaptive and less sensitive to manual configuration.

In summary, ProTrader-RL (2024) represents an important evolution and milestone, it really aligns closely with my project's goal of building a transparent, adaptable, and multi-agent financial advisor.

2.6 Parallel Multi-Module Reinforcement Learning for Stock Trading

Another study that caught my attention was the *Parallel Multi-Module Reinforcement Learning Algorithm for Stock Trading* (2024), which provides an elegant and scalable interpretation of reinforcement learning applied to financial markets.
The authors formulate stock trading as a **Markov Decision Process (MDP)**, in which an agent interacts with the market by buying or selling assets and receiving rewards based on portfolio performance. The overall objective is to **maximise the expected**

cumulative discounted return, the same principle underpinning the reinforcement-learning theory studied during the AI module.
In fact, this framework immediately reminded me of the **DQN Agent** I implemented during the course. Both systems are grounded in the same theoretical foundation: the **Q-value function**, which estimates the long-term expected reward of taking action while training the Atari Breakout agent.

One difference comparing to the agent we have been using during the course is that this approach uses the **Double Deep Q-Network (DDQN)** algorithm: each agent is trained independently for multiple episodes (using 75 % of the dataset for training and 25 % for testing) with realistic transaction fees of 0.1 % for buys and 0.2 % for sells.

What makes this work particularly relevant is its **parallel multi-module architecture**: rather than relying on a single network, the authors run several DDQN agents in parallel, each focusing on a **specific stock** or trading subtask. This improves efficiency, increases exploration coverage, and reduces variance between runs. The group of agents is then aggregated into a unified global policy, producing more stable overall performance.

However, the authors recognise several **limitations** that are important to note:

1. The approach depends heavily on access to reliable fundamental data, which is unavailable for many assets such as cryptocurrencies.

2. It was tested only on **single-stock trading**, so extending it to multi-asset portfolio management remains an open research direction.

3. Explicit **risk management** components are not integrated into the RL process, meaning the agent may overfit to short-term profit maximisation.

The study offers important insights for building large-scale RL systems: parallel learners can collectively produce more stable global policies, reflecting a closer approximation of real market dynamics than single-agent approaches.

2.7 Literature considerations

The last two articles I have been studying, *ProTrader-RL (2024)* and the *Parallel Multi-Module RL* framework, have been very important and useful to me: I could really see the connections with what we studied and how they try to improve some concepts (such as the double DQN or the global policy). Also, comparing to the other researches, I think they highlight the advantages of decomposing trading into modular decision units, and of running agents in parallel to accelerate learning and improve robustness.

My project builds on these two complementary ideas. It aims to integrate a **multi-**

agent RL architecture inspired by ProTrader-RL, enhanced with **parallel processing** to improve exploration efficiency, and extended with **Genetic Algorithms** to evolve hyperparameters and reward functions automatically. Additionally, I will incorporate a **sentiment-analysis agent** and a **Natural-Language layer** to make the system not only adaptive and data-driven but also transparent and human-understandable.

3 Design

Word Count: 1731

This project designs and implements an AI-driven Financial Advisor Bot that generates buy/sell recommendations from historical market data using a modular reinforcement learning (RL) pipeline. The core decision-making is performed by two specialised Double Deep Q-Network (DDQN) agents (Buy and Sell). A deterministic coordination layer (TradeManager) enforces realistic trading constraints and ensures interpretable control flow. On top of the trained agents, I used Genetic Algorithm (GA) as an outer-loop optimiser to tune key decision thresholds and execution parameters (such as buy confidence threshold) in order to improve out-of-sample performance. Finally, I developed a lightweight API and NLP-style response layer to expose recommendations in a user-facing format.

The design prioritises modularity and extensibility, reproducibility and controlled evaluation, and interpretability through explicit separation of learning and rule-based execution.

3.1 Project overview

Inspired by the ProTrader-RL concept, the system emulates a simplified professional trading workflow:

data preparation -> entry decision -> exit decision -> execution constraints -> evaluation -> user-facing explanation.

Rather than implementing a single monolithic agent, the architecture I designed aims to separate responsibilities into distinct components:

- **Data processing** produces aligned market-state vectors.
- **Buy agent** learns entry timing (BUY vs HOLD).
- **Sell agent** learns exit timing (SELL vs HOLD), conditioned on an open position.
- **TradeManager** coordinates lifecycle and enforces constraints (one position at a time, holding rules, transaction costs, forced exits).
- **Genetic Algorithm** tunes decision/execution parameters on a validation set.
- **Decision API** serves predictions and confidence metrics through a web UI.

This separation supports targeted experimentation (e.g., replacing indicators, changing reward shaping, or extending the GA search space) without destabilising the rest of the system.

3.2 Domain and Users

The domain is algorithmic trading and financial decision support, which naturally demands adaptability and interpretability. Financial markets are non-stationary: volatility, sentiment, and macroeconomic changes can render static models obsolete. RL provides adaptability, while the NLP layer supports interpretability, two essential requirements in finance.

Although implemented as a research prototype, the system targets three user groups:

- **Retail investors**, who need understandable recommendations and simple explanations.
- **Researchers/students**, who need modular evaluation of RL training, reward design, and optimisation.
- **Developers**, who can integrate the inference API into broader fintech prototypes.

The design balances adaptability (learning agents), safety and validity (TradeManager constraints), and transparency (exposed confidence/Q diagnostics and natural-language summaries).

3.3 System Architecture

The overall architecture of the *Financial Advisor Bot* follows a **multi-agent reinforcement-learning framework** inspired by *ProTrader-RL (2024)* and extended with genetic hyperparameter evolution, and natural-language interpretability.
As mentioned above, the system emulates the workflow of professional traders, with distinct agents specialising in data preparation, decision-making, and validation.

Each agent operates within a unified environment coordinated by a deterministic TradeManager, while a Genetic Algorithm (GA) optimises execution-level parameters across generations.

At a high level, the system comprises **five functional layers**, as illustrated in the following diagram

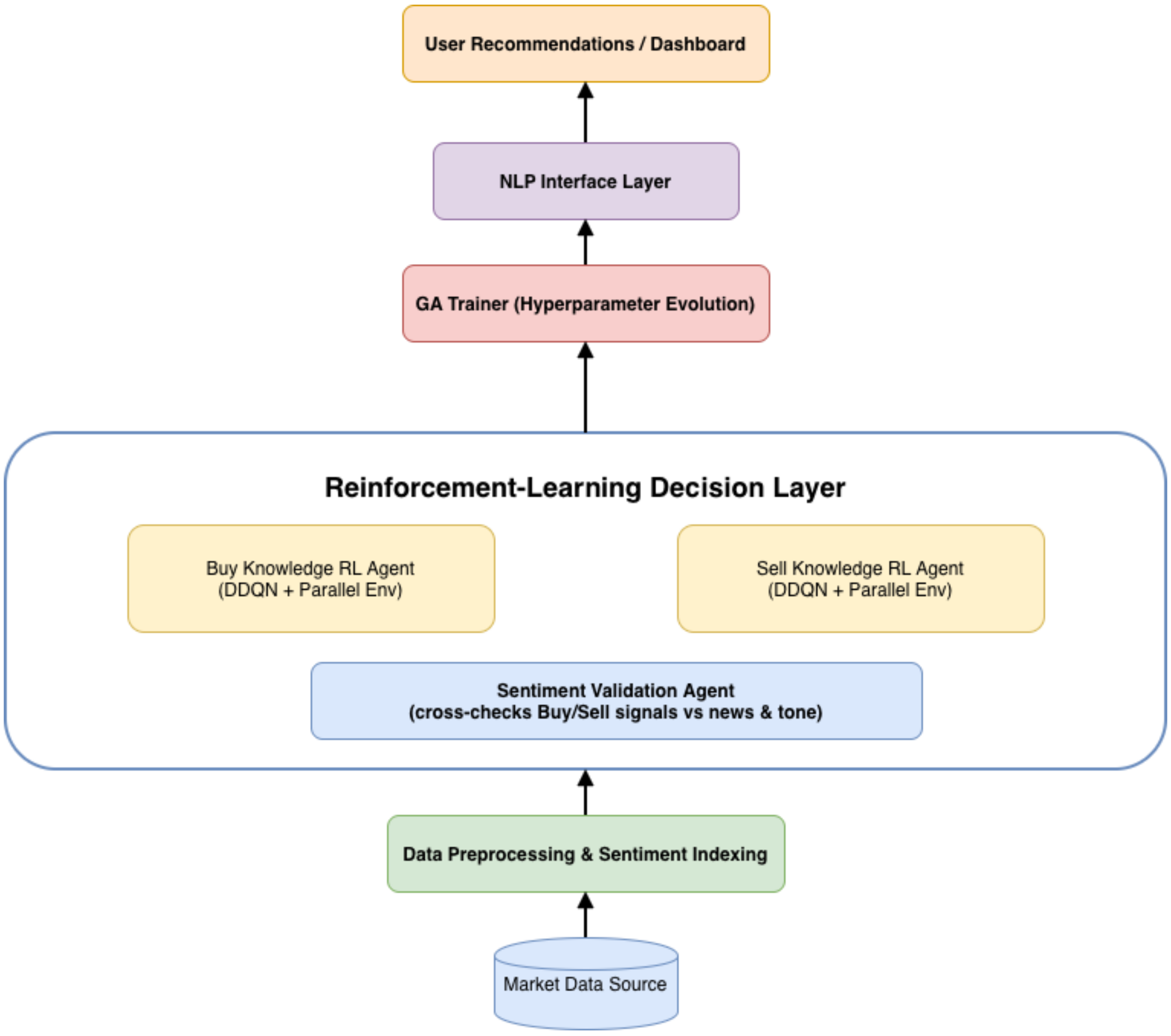

User Recommendations / Dashboard
NLP Interface Layer
GA Trainer (Hyperparameter Evolution)
Reinforcement-Learning Decision Layer
Buy Knowledge RL Agent
(DDQN + Parallel Env)
Sell Knowledge RL Agent
(DDQN + Parallel Env)
Sentiment Validation Agent
(cross-checks Buy/Sell signals vs news & tone)
Data Preprocessing & Sentiment Indexing
Market Data Source

Figure 3: Diagram about project architecture

Layer 1 - Data Preprocessing and Sentiment Indexing

This layer constructs time-aligned state representations from historical market data using a financial provider service (Yahoo Finance) and transformed into a feature matrix that includes prices, returns, and a compact set of technical indicators. The pipeline aligns features across tickers and dates and exports a consistent dataset used for both training and inference.

I also added a sentiment signal as an additional bounded feature: sentiment is derived from the Alpha Vantage News & Sentiment API, which provides daily polarity and relevance (“mass”) values. The output of this layer is a deterministic state representation suitable for RL environments.

Output: a time-indexed feature tensor and aligned metadata (ticker, date).

Layer 2 - Reinforcement-Learning Decision Layer

The decision layer contains two trained agents using the same DDQN architecture but operating under different assumptions:

Buy Knowledge Agent (DDQN entry policy) learns when to open a position using actions {BUY, HOLD}. The agent observes the current market state and estimates Q-values for each action, selecting the best expected long-term outcome.

Sell Knowledge Agent (DDQN exit policy) learns when to close an existing position using actions {SELL, HOLD}. The sell environment is conditioned on a prior entry and can include additional position-related features (e.g., time in position, unrealised return). This reflects the asymmetric nature of entry vs exit decisions and allows the sell policy to optimise profit-taking and risk control.

Further details related to the RL decision layer are available in the section 3.4.2

Layer 3 - TradeManager Coordination + GA Optimisation

I developed the **TradeManager** component in order to keep trade execution logic deterministic. The TradeManager is responsible for coordinating Buy/Sell agent outputs and enforcing trading constraints, such as:

- one position at a time
- minimum holding period and cooldown between entries
- transaction cost handling
- forced exits (time-based or horizon-based)
- confidence gating and margin constraints

This design aims to improve interpretability: each executed decision is explainable as “agent signal + explicit constraint”.

Genetic Algorithm (outer-loop optimisation)
The GA operates on top of the agents to tune TradeManager parameters and decision thresholds that impact realised performance (e.g., minimum buy confidence or sell margin thresholds). Importantly, in this implementation the GA does **not** modify neural-network weights. Instead, it searches over configuration parameters using validation performance as a fitness signal, then evaluates the best genome once on a held-out test set.

Layer 4 - Natural-Language Interface

I developed a FastAPI-based service to expose the trained models through an API and a lightweight web UI. The *DecisionEngine* loads saved PyTorch checkpoints and produces a structured response including:

- action recommendation (BUY/SELL/HOLD)
- confidence score derived from Q-values
- Q-gap (difference between top actions)
- date/ticker metadata

An NLP-style response layer converts these metrics into short user-facing explanations. This layer does not influence decisions; it exists to improve transparency and usability. The system exposes both the Q-value gap (difference between the highest and second-highest action values) and a derived confidence score. The Q-gap is a raw diagnostic measure, while the confidence score is a normalised indicator used for decision gating and UI reporting. This separation improves interpretability without affecting the trained policy.

3.4 Key Technologies and Methods

The technologies used in this project follow directly from the methods highlighted in the literature review. **Reinforcement Learning** (RL), Deep Q-Networks (DQN), and **Genetic Algorithms** (GA) form the core methodological foundation, with the design adopting modern enhancements such as **Double DQN (DDQN**), modular RL architectures, and evolutionary hyperparameter optimisation.

3.4.1 Data Preprocessing & Sentiment Indexing

I did fetch the market data from standard financial data sources (Yahoo Finance) and enhanced it with a compact set of technical indicators commonly used in RL-based trading research. These features are cleaned, aligned, and assembled into fixed-length state windows suitable for RL. A simple sentiment index using sentiment model (AlphaVantage) provides additional context. This mirrors recent work that integrates market sentiment into trading environments to improve decision robustness.

3.4.2 Reinforcement-Learning Decision Layer

The Buy and Sell Knowledge Agents are implemented using **Double Deep Q-Networks (DDQN**) within custom **OpenAI** Gym environments. In this setting, the agent learns a *Q-function*, which estimates how good each possible action is in a given market state based on expected future outcomes.

In practice, the Q-function is approximated by a neural network. For a given market state, the network outputs one Q-value per available action. For example:

Q(state, BUY) = 0.62
Q(state, HOLD) = 0.45

This means that, based on past experience, buying is expected to lead to better long-term results than holding in similar situations. The Q-values themselves are not decisions but estimates that guide the policy. In this project, the policy is derived from these Q-values and further filtered using confidence thresholds to reduce noisy or low-conviction trades.

Double DQN is used to reduce Q-value overestimation, a well-known issue in standard DQN methods, which is particularly problematic in noisy financial environments.

Rewards are defined using risk-adjusted formulations inspired by direct-reinforcement approaches and modern financial RL frameworks such as FinRL and ProTrader-RL.

Neural network → estimates the Q-function → outputs Q-values → policy decides what to do (with confidence gating).

3.4.3 TradeManager and Decision Coordination

The Buy and Sell Knowledge Agents learn specific policies, but the system requires a deterministic coordination layer to ensure that actions are executed in a valid and consistent sequence. For this reason, I designed a TradeManager component that acts as the central controller of the trading lifecycle.

The TradeManager enforces structural constraints such as:

- A Buy action can be executed only when no position is open
- A Sell decision can be executed only after an entry has occurred
- Risk and execution rules (e.g., maximum holding time, confidence thresholds, and transaction cost handling) are applied consistently across agents.

The TradeManager does not learn: instead, it provides rule-based control flow that stabilises multi-agent interaction, reduces conflicting decisions, and improves interpretability by ensuring that each decision can be linked to a specific module and constraint.

3.4.4 Genetic Algorithm

The Genetic Algorithm (GA) is implemented as an outer-loop optimiser operating over execution-level parameters rather than neural network weights. Each GA individual

represents a specific configuration of TradeManager thresholds and constraints (such as buy confidence or sell margin thresholds).
These configurations are evaluated on a validation split using realised portfolio performance as a fitness signal. The best-performing genome is then evaluated once on a held-out test set to assess out-of-sample generalisation.
The GA does not modify DDQN weights; it optimises the deterministic execution layer that converts agent outputs into trades.

The GA fitness score is computed on the validation split only. The held-out test split is used strictly for final evaluation of the best-performing genome after optimisation. This separation ensures that hyperparameter selection does not leak information from the final evaluation dataset.

3.4.5 NLP Explanation Layer

A lightweight NLP component translates model outputs into short, human-readable sentences. This supports transparency and aligns with the project's goal of combining adaptive RL behaviour with clear financial recommendations.

"The system suggests opening a position in Apple this week, as technical momentum and market sentiment remain positive with moderate risk."

This layer does not influence decision-making, its only scope is to improve transparency, trust, and usability for non-technical users.

3.4.6 User Interface

The interface exposes both the decision and supporting diagnostics (confidence and q-gap), enabling users to interpret the recommendation rather than receiving a black-box output.

3.5 Work plan

Because this is the final report, the project plan is summarised as a completed development process rather than a forward-looking schedule. The project followed a structured, agile-driven development plan designed to manage the complexity and risk related to a multi-agent reinforcement-learning system.

TABLE 1: COMPONENT DEPENDENCIES AND MAIN SCOPE.

Component	Dependency	Description
Data preprocessing	Market data APIs, feature engineering	Time-aligned state representation
Buy Agent	Feature pipeline, state representation	Core learning component used to validate RL feasibility before adding system complexity
Sell Agent	Stable Buy entries, Trade Manager	Exit rewards must be evaluated relative to valid entry points to avoid noisy learning signals
Trade Manager	Buy and Sell agents	Enforces trading lifecycle constraints and ensures interpretable, deterministic execution
Sentiment Filtering	Buy Agent outputs	Acts as a risk-control and validation layer rather than a primary trading signal
Genetic Algorithm	Trained agents and Trade Manager	Optimises decision thresholds without destabilising learned policies
NLP Interface	Stable decision outputs	Introduced after core learning stabilisation to avoid masking decision errors

4 Implementation

Word Count: 1913

This section describes how I implemented the system architecture defined in the Design section. The implementation focuses on four core components: a feature pipeline for state construction, a Buy Agent for entry decisions, a Sell Agent for exit decisions, and a Trade Manager that orchestrates execution, accounting, and lifecycle constraints. These components interact through well-defined interfaces to ensure reproducible training and evaluation.

4.1 Project Architecture overview

I have been following the design described above and I have been working on these four core components:

- **Feature Pipeline**: transforms historical price data into fixed-length state representations.
- **Buy Agent**: a reinforcement learning agent responsible for deciding when to open a long position.
- **Sell Agent**: a separate reinforcement learning agent trained to decide whether to hold or close an open position.
- **Trade Manager**: a backtesting engine that orchestrates trade execution, applies transaction costs, and enforces time and segment constraints.

The Buy Agent and Sell Agent operate independently but interact through the Trade Manager, which ensures that only one position can be open at any time and that all exits follow the same accounting rules.

To support training across multiple financial instruments, the dataset is split into fixed-length segments, each corresponding to a single ticker (e.g. AAPL, MSFT, NVDA). Segment boundaries are strictly enforced to prevent information leakage and ensure that all trade lifecycles remain confined to a single instrument.

4.1.1 Development Iterations and Design Decisions

The prototype I have been working on for the mid-term assessment focused on a single reinforcement learning agent (the buy agent) responsible only for trade entry decisions. This simplified setup was used to validate the feature pipeline, backtesting infrastructure, and confidence-based decision logic before introducing additional complexity.

As originally designed, I added a dedicated Sell Agent to allow early trade exits. During the first implementation of this sell agent, sell decisions were managed as independent exit points. However, this approach was very ineffective: I noticed that exit decisions that were not explicitly conditioned on the original buy entry resulted in highly noisy reward signals, making it difficult for the agent to learn a stable policy.

This observation highlighted an important design constraint: sell decisions must be evaluated **relative to the entry that generated the position**, rather than as isolated actions. As a result, I redesigned the Sell Environment so that each episode begins at a valid buy entry index, ensuring that exit rewards are meaningfully aligned with the corresponding trade.

Another challenge I was facing during the first experiments is that the learning progress for the Sell Agent remained limited due to the relatively small number of available entry points. To address this, I started collecting more high-quality buy candidates across each data segment: this change increased the number of Sell Agent training episodes and improved sample efficiency.

4.2 Features and Indicators

The state representation combines price-derived indicators and sentiment signals.

Table 2: indicators included in the final feature vector.

Indicator	Description	Purpose
Close Return	Daily percentage return	Captures short-term momentum
Rolling Mean (n)	Moving average of close price	Trend smoothing
Rolling Std (n)	Rolling volatility	Risk estimation
RSI	Relative Strength Index	Overbought/oversold detection
MACD	Moving Average Convergence Divergence	Trend change detection
Sentiment Score	Normalised news polarity	Market mood
Sentiment Index	Relevance-weighted news intensity	Signal strength

4.3 Buy Agent and Entry Logic

The Buy Agent is trained to choose between two actions: **HOLD** and **BUY**. Rather than acting directly on raw Q-values, the agent's output is transformed into a **confidence score**, derived from the relative separation between action values. I open a position only when this confidence exceeds a configurable threshold defined in the system configuration and later optimised via a Genetic Algorithm.

```
q_buy = np.asarray(self.buy_agent.q_values(s), dtype=np.float32)
if conf >= BUY_MIN_CONF:
    self._open(t, price, meta={"buy_conf": conf})
```

A trade is opened only when this confidence exceeds a configurable threshold, introducing a margin-based decision mechanism that filters out uncertain or low-conviction signals. This design improves robustness across market regimes and mirrors human decision-making practices, where actions are taken only when perceived advantage is sufficiently clear.

Once a position is opened, the Trade Manager records the entry index and price, applies entry transaction costs, and transfers control to the Sell Agent for subsequent decision-making.

4.3.1 Sentiment Integration in Buy Logic

In addition to price-based features, I developed a news-based sentiment signal to provide contextual information at trade entry time. Sentiment is applied **only to the Buy decision** and does not affect Sell Agent behaviour.

I used the Alpha Vantage News & Sentiment endpoint, which provides a daily pre-computed sentiment score in the range [−1,1] together with a sentiment “mass” representing the relevance and volume of contributing news items. These values are aligned with the daily price timeline and I appended them to the feature set. When sentiment data is unavailable, I use a neutral default value to preserve sequence length consistency.

Before opening a position, the Trade Manager evaluates a configurable sentiment acceptance rule. Trades are blocked if sentiment is sufficiently negative or weakly supported by low mass, even when the Buy Agent’s confidence threshold is exceeded. This mechanism acts as a soft risk filter, reducing exposure during unfavourable news conditions while preserving a fully price-driven exit strategy.
All sentiment thresholds are defined in the configuration file and can be enabled, disabled, or later optimised via Genetic Algorithms.

4.3.2 Bellman equation and related challenges

Both Buy and Sell agents are trained using Double Deep Q-Networks, which rely on Bellman-based temporal-difference updates like the Atari game agent we have been studying. However, financial time series are noisy and non-stationary, which can amplify instability in long-horizon bootstrapped value estimates. To mitigate this, I integrated bounded decision horizons, delta-based reward formulations (particularly for the Sell Agent), and post-processing of Q-values via confidence gating.

This design is based on the earlier direct-reinforcement approaches in finance, such as Moody and Saffell (2001), which prioritise stable, risk-aware optimisation over theoretical value convergence.

4.4 Trade Manager

The Trade Manager acts as the central coordinator of the system, enforcing trading constraints, applying transaction costs, and logging completed trades. Its

responsibilities include enforcing a single open position, applying consistent transaction costs, enforcing time-based exits, and respecting segment boundaries when working with multi-asset datasets.
All training and evaluation runs are executed through a specific pipeline script that produces structured logs, including trade, entry indices, and summary statistics in JSON format. Examples of these logs are available in the Evaluation section.

This design ensures that learning, evaluation, and later optimisation phases operate under identical assumptions, improving reproducibility and comparability across experiments.

4.5 Sell Agent: Environment, Reward, and Integration

To train the Sell Agent, I implemented a dedicated reinforcement-learning environment in which each episode begins at a valid Buy entry index. This design ensures that exit decisions are evaluated relative to the specific trade that generated the position, rather than as isolated actions. The agent observes the same market features used by the Buy Agent, augmented with position-specific information.

The Sell Agent's observation logic includes the unrealised return since entry, the fraction of the allowed holding period already elapsed, and the remaining holding time: this allows the agent to condition its decisions on both market context and the trade lifecycle, improving temporal awareness and exit timing.

Rather than optimising absolute profit, I trained the Sell Agent using a delta-based reward formulation. The reward measures the improvement achieved by selling at the current timestep compared to holding the position until the forced exit at the horizon limit. Intermediate HOLD actions receive zero reward, while terminal rewards are issued only when the position is closed or forcibly exited. This encourages exits only when they meaningfully improve outcomes relative to passive holding.

Transaction costs are applied multiplicatively to equity updates during entry and exit events, ensuring consistent accounting across training and evaluation.

During backtesting, the Sell Agent is queried only after a minimum holding period has elapsed. Similar to the Buy Agent, exit decisions are subject to a confidence threshold to prevent marginal or unstable exits. I close a position only when the Sell action is selected with sufficient confidence and no forced exit condition applies.
The Sell Agent produces a combination of voluntary exits and horizon-based closures, demonstrating active participation in trade management rather than defaulting to time-based exits.

For example, in one evaluation run the Sell Agent produced 64 voluntary exits and 34 horizon-based exits, with an average holding duration of 15.9 bars.

These behaviours are logged during training and evaluation runs and are analysed further in the Evaluation section.

4.6 Genetic Algorithm for Hyperparameter Optimisation

To improve the tuning of decision thresholds and trading constraints, I implemented a Genetic Algorithm (GA) component that optimises selected configuration parameters without modifying the core training code.

The GA operates exclusively on execution-level parameters: each individual (genome) represents a set of tunable "genes" (e.g., *buy_min_confidence, sell_min_margin, sell_min_delta_vs_hold, min_hold_bars, cooldown_steps*). For each genome, the system re-runs deterministic backtests using fixed pre-trained Buy and Sell agents. Fitness is computed from realised portfolio performance on a validation split. The best genome is stored (best.json) and loaded during inference, ensuring that deployment reflects the optimised configuration without retraining neural networks.

I created the GA component in a way that can easily scale from a single parameter to multiple parameters by extending the gene specification list, without changing the evaluation loop.

4.7 End-to-End System Architecture

I developed the project as a modular pipeline that separates data processing, decision-making, optimisation, and user interaction, relying on pre-trained reinforcement learning agents (Buy Agent and Sell Agent) coordinated by the Trade Manager, with configuration parameters optimised via a Genetic Algorithm.

I load the trained agents together with GA-optimised parameters and I apply them to the latest market data and sentiment signals. The Trade Manager orchestrates all decisions and produces structured outputs, including action recommendations, confidence scores, sentiment status, and decision metadata. These outputs are designed to be consumed by higher-level services without requiring direct access to the learning components.

At inference time, a dedicated DecisionEngine loads the trained PyTorch checkpoints via a ModelAdapter wrapper and exposes structured outputs through a Decision dataclass, including action, confidence, Q-gap, and metadata. This abstraction layer decouples model loading from user-facing services and ensures consistent evaluation across offline and deployed settings.

This separation allows the learning pipeline, optimisation process, and user-facing interfaces to evolve independently while preserving consistent execution logic and reproducibility.

4.8 Natural Language Interface

The NLP component purely acts as an interpretation and routing layer rather than a decision-making module. Its primary responsibility is to translate user queries into structured requests that can be evaluated by the trading pipeline. This includes intent classification (e.g. buy, sell, analysis), entity extraction (e.g. company name or ticker symbol), and temporal or contextual information when present.

Once a query is parsed, the NLP layer invokes the pipeline using the most recent market data, sentiment signals, and GA-optimised configuration parameters. The response returned to the user is the outputs of the Buy Agent, Sell Agent, and Trade Manager, enhanced with explanatory metadata such as confidence scores, sentiment filters, and recent trade behaviour.

4.9 Web Interface and API interface

To make the system accessible to end users, the architecture includes a web interface built on top of a dedicated API layer. The API exposes a small number of endpoints responsible for handling requests, configuration inspection, and historical result retrieval.

The web interface is designed to provide an intuitive user experience, allowing users to submit investment-related questions, inspect current recommendations, and explore supporting information such as recent sentiment trends or historical trade statistics. Rather than exposing raw model internals, the interface shows summarised outputs aligned with the user's query and risk profile.

From an architectural perspective, the web interface and NLP layer are fully decoupled from the training and optimisation pipelines. Reinforcement learning training, sentiment ingestion, and GA-based parameter optimisation are executed offline, while the deployed system relies exclusively on pre-trained models and configuration artefacts.

The web interface acts purely as a thin client over the inference API and does not influence model training, evaluation, or optimisation results.

5 Evaluation

Word Count: 2198

This section evaluates the current implementation as a *working preliminary system* and focuses on verifying correctness, stability, and measurable outcome from each architectural component.

The evaluation is intentionally incremental: I started testing each component (Buy Agent, Sentiment index, Sell Agent, Trade Manager integration, GA optimisation) in isolation and then in the full end-to-end pipeline.

5.1 Evaluation methodology and basic setup

The evaluation strategy I adopted in this project is explicitly aligned with its primary aim: validating the correctness, stability, and extensibility of a modular reinforcement-learning–based financial decision system, rather than, at this stage, maximising short-term trading profitability. For this reason, I did focus on component-level validation, learning dynamics, and relative performance improvements between architectural variants under identical historical conditions. I wanted to prioritise interpretability, reproducibility, and robustness, which are essential for a decision-support system intended to operate in non-stationary and noisy financial environments.

I created a multi-asset dataset by concatenating fixed-length segments, where each segment corresponds to one ticker (segment boundaries are handled as hard constraints to avoid cross-asset information leakage and to ensure that each trade lifecycle remains within a single ticker). For each segment, I did split the data into train and test sets using a fixed train segment length, then stacked back into train and test sets. This produces two aligned arrays: a feature matrix and a price vector.

Execution and reproducibility

I ran all the experiments through dedicated scripts:

- *build_features.py* generates feature arrays (including sentiment index/score alignment) and exports them as NumPy files.
- *run_pipeline.py* trains agents, produces trade logs, and generates structured summaries (summary.json, trades_*.json, entry index arrays).
- *run_ga.py* evaluates candidate genomes and logs GA outputs (*ga_log.json*l, *best.json, meta.json*).
- *Plot_results.py* reads the content of specific folders, looking for the generated JSON outputs and *npy* files) and plot the results

Each run is associated with a unique run directory and fixed random seed. This ensures experiments can be reproduced exactly and compared across iterations: reinforcement learning outcomes may vary across seeds due to stochastic exploration and replay sampling (future work might include evaluating variance across multiple seeds to quantify stability).
In addition to the automated pipeline scripts, I also developed Jupyter notebooks and diagnostic scripts to inspect intermediate tensors, reward calculations, and trade-level

accounting step by step. These tools were very helpful to me for verifying correctness of reward propagation, entry harvesting for Sell training, and transaction cost application.

Evaluation metrics

Performance is measured using:

- **Final equity** (equity result after transaction costs)
- **Average net return per trade**
- **Win rate**
- **Trade count and exit reason distribution** (forced exits vs Sell Agent exits)

Also, diagnostic metrics are logged to validate decision gating:

- Buy entry debug count (checked, opened, blocked by trend, blocked by confidence, blocked by sentiment, etc.)
- Sell decision debug count

These metrics are reported separately for train and test to identify overfitting and generalisation gaps. Future evaluations might include risk-adjusted metrics such as maximum drawdown and volatility-adjusted return (e.g., Sharpe ratio) to better assess downside risk.

5.2 Feature Scaling Validation and Stability Analysis

This subsection evaluates the impact of the feature scaling and normalisation strategy on learning stability and correctness.
All price-based features used by the Buy and Sell Agents are normalised using rolling window statistics rather than global scaling.Specifically, each feature is transformed using rolling normalisation techniques (e.g. rolling z-score or rolling min–max scaling), computed over a fixed historical window before the current timestep. This ensures that feature values remain comparable across time while avoiding look-ahead bias, as only past information is used at each decision point.

This design choice provides several benefits. First, it makes the learning process stable by keeping feature values within consistent ranges, improving gradient behaviour in the DDQN networks. Second, it allows the agents to adapt more effectively to regime changes, such as shifts in volatility or trend strength, without requiring retraining on the entire historical dataset. Also, it reduces the risk of bad / false correlations caused by long-term price drift dominating feature values.

The sentiment features are handled separately because sentiment scores are already in the range [−1, 1], hence no additional scaling is applied (when sentiment data is missing for a given timestep, I apply a neutral default value to preserve the sequence continuity and avoid introducing artificial discontinuities in the feature space).

Overall, the combination of rolling feature scaling and bounded sentiment inputs contributes to improved learning stability and robustness, particularly when training agents across multiple assets and extended historical periods.

5.3 Validation Strategy and Data Splitting

Financial reinforcement learning presents unique challenges due to temporal dependencies and non-independent observations. For example, classical k-fold cross-validation we have been using during the ML course, is inappropriate in this context because it violates time ordering and introduces information leakage.

In this project, validation is performed using a deterministic, segment-based train/test split that preserves temporal causality. Each financial data is divided into fixed-length segments (tickers), with an initial portion allocated to training and the remaining portion reserved for testing. Segment boundaries are strictly enforced during both training and evaluation to prevent trades from crossing between instruments or time regimes.
This approach ensures that all evaluation results fairly match out-of-sample behaviour. I'm aware that this strategy does not fully capture overall generalisation, however it provides a leakage-free baseline suitable for validating system correctness, learning stability, and relative improvements between configurations.

More advanced validation techniques are planned for future work in order to try to reflect real-world deployment scenarios by repeatedly retraining and evaluating the system over sequential time windows. However, for the preliminary assessment, the current validation strategy is sufficient to demonstrate the correctness and robustness of the proposed architecture and developed pipeline.

5.4 Overfitting Risk and Evaluation Limitations

A well-known challenge in financial backtesting is overfitting. High returns / rewards *in*-sample do not necessarily translate to robust *out-of*-sample behaviour, specially in noisy environments.
I tried to apply several design choices to mitigate and reduce overfitting risk. These include strict train/test separation, enforced segment boundaries, rolling feature normalisation, confidence-based trade filtering, and delta-based Sell Agent rewards that discourage trivial optimisation. Also, all experiments are fully reproducible through fixed random seeds and structured logging.
Nevertheless, the current evaluation remains limited to historical simulation. I did not conduct human user studies at this stage on purpose, as the system is intended to function as a decision-support engine rather than an interactive trading interface. My priority is the validation of the stability and consistency of the underlying decision logic.

5.5 Buy Agent Evaluation and Learning Behaviour

This subsection evaluates the Buy Agent and the entry filters that determine when the system opens a position. The Buy Agent is trained as a two-action DDQN (HOLD vs BUY), but trades are not opened directly from argmax actions. Instead, Q-values are converted into a confidence score and gated by a configurable threshold (subjected to GA optimisation). This design aims to reduce noisy entries and improve decision robustness.

The training loop logs episode rewards, epsilon decay, loss values, and target-network synchronisation events. These logs are used to confirm that learning is progressing and that the agent is not stuck in a specific policy (e.g., never buying). Training stability is assessed via decreasing loss and consistent reward behaviour rather than absolute reward.

During Trade Manager execution, every candidate timestep is counted and categorised into entry outcomes:

- blocked by trend filter
- blocked by “latest entry” constraint (ensuring the trade can complete within horizon/segment)
- blocked by confidence (below *buy_min_confidence* - later subjected to GA)
- blocked by sentiment
- opened trades

This produces an auditable breakdown of why entries were accepted or rejected. In the current runs, sentiment filtering blocks a small subset of otherwise-valid entries, and the pipeline records representative blocked samples including the timestep, sentiment score, sentiment mass, and buy confidence. This supports verification that sentiment is functioning as a *soft risk filter* rather than dominating the trading behaviour.

5.6 Sell Agent Evaluation, Integration and Learning Behaviour

This subsection evaluates the Sell Agent and its integration into the Trade Manager. My main goal was to demonstrate the improvements when buy and sell agents “work together” (instead of the early exits outcomes relative to the Buy-only baseline).
The Sell Agent is trained in a dedicated Sell Environment where each episode begins at a valid buy entry index. Observations consist of the same base market features plus position context (e.g. unrealised return or time elapsed fraction). The reward is delta-based: the terminal reward measures whether selling now improves net return compared to holding until the forced exit. This reward structure aligns Sell decisions to the originating entry and avoids the noisy reward problem observed in early prototypes.

Buy-only vs With-sell comparison

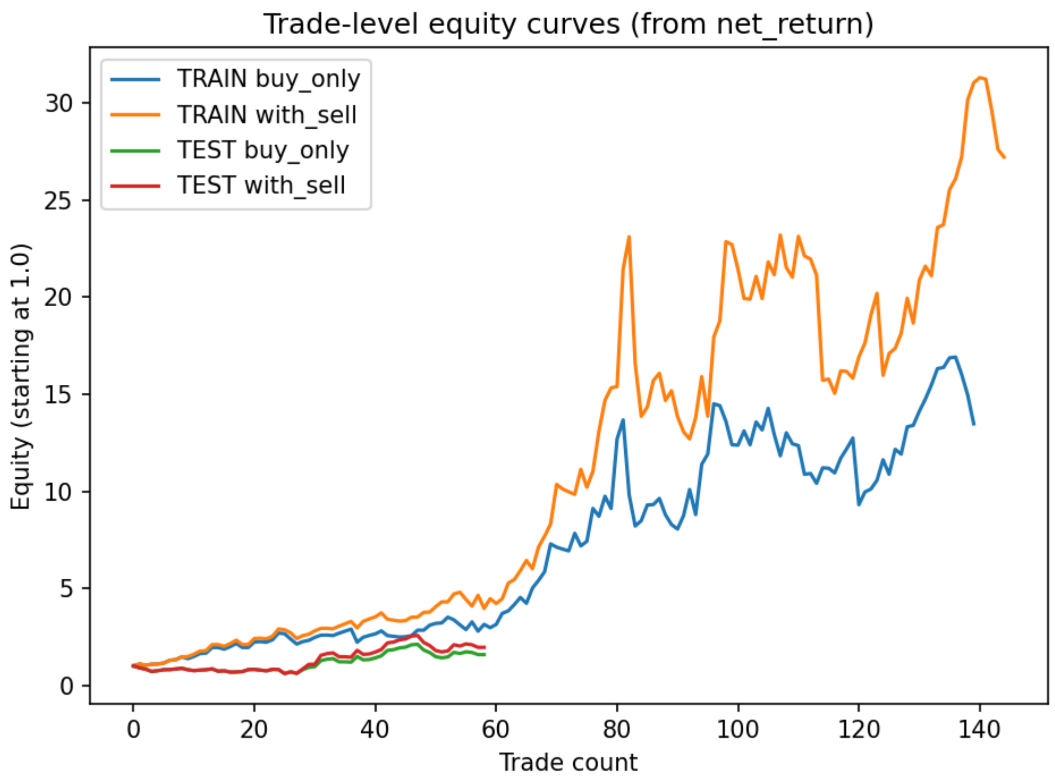
To verify that the Sell Agent contributes positively at the system level, I did evaluate the same trained Buy Agent under two conditions:

1. Trade Manager Buy-only: all exits occur at the forced horizon

2. Trade Manager With-sell: Sell Agent may exit early (after a minimum hold) if gating conditions are met

The output logs provide an explicit report ("WITH_SELL – BUY_ONLY"), including final equity delta, average net return delta, trade count delta, and number of Sell Agent exits. This was a key evidence and milestone for my project, showing that Sell integration is operational and beneficial beyond training-only metrics.

Over training episodes, the agent progressively increases the proportion of voluntary exits relative to forced exits, indicating that it learns to identify situations where early exit improves realised outcomes rather than defaulting to time-based closure.



*Cumulative **trade-level equity curves** derived from per-trade net returns*

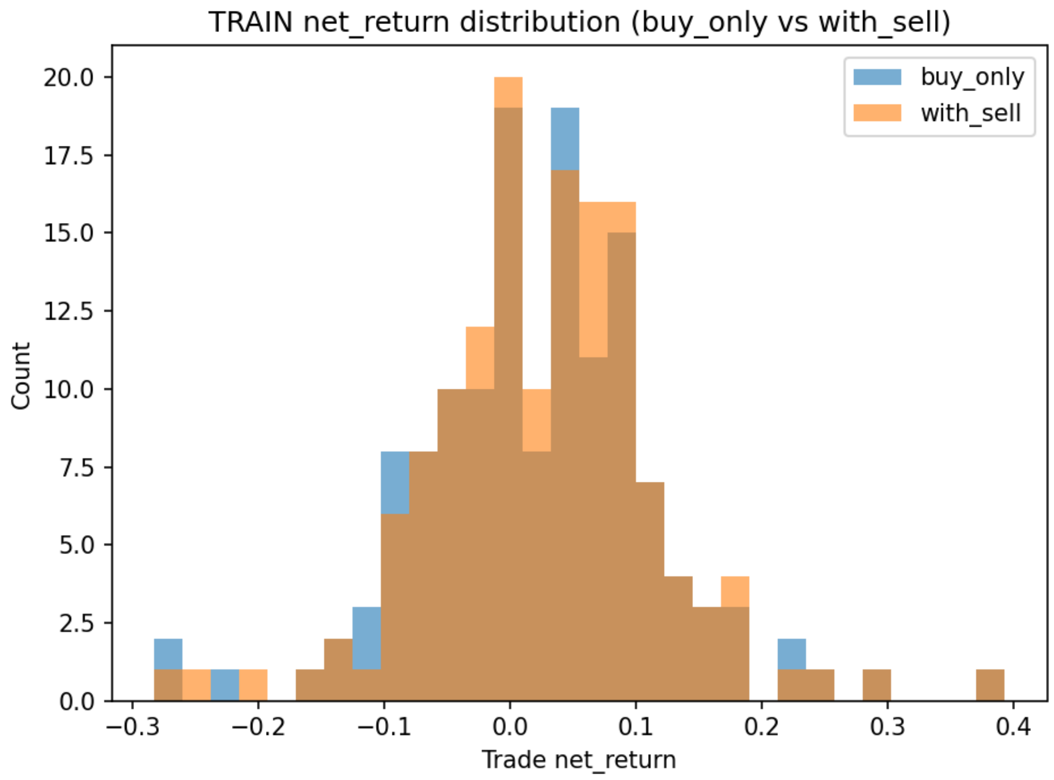


*Figure 5: Net-return distributions on the **training set***

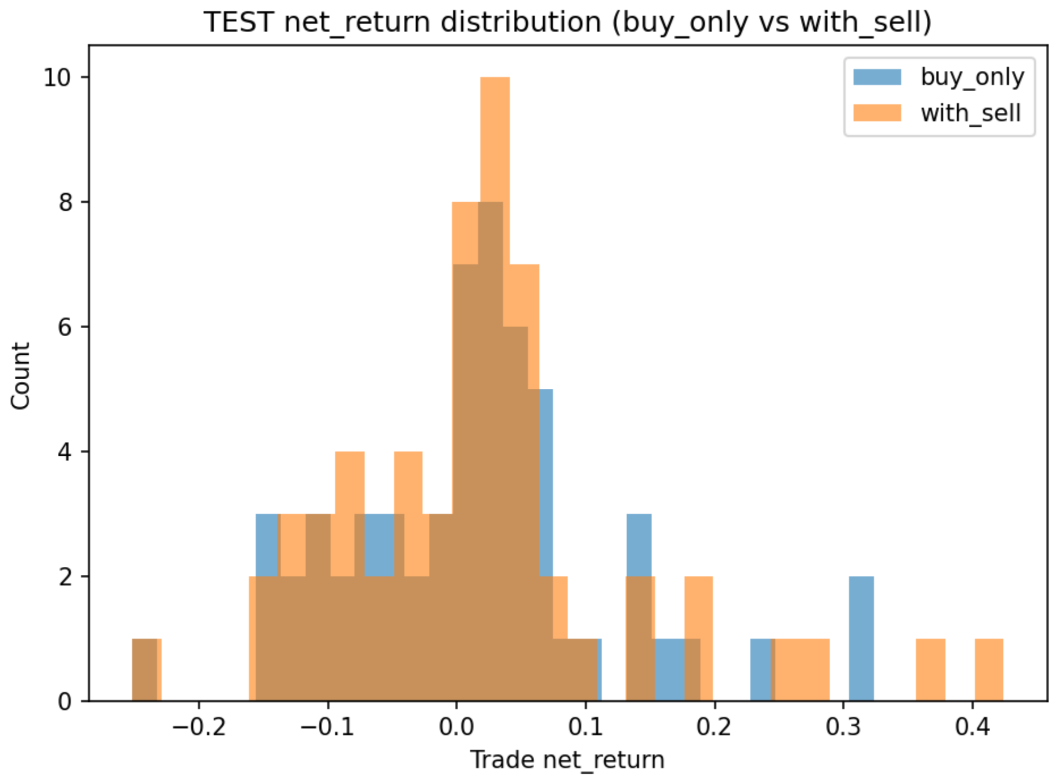


Figure 6: Net-return distributions on the ***test set***

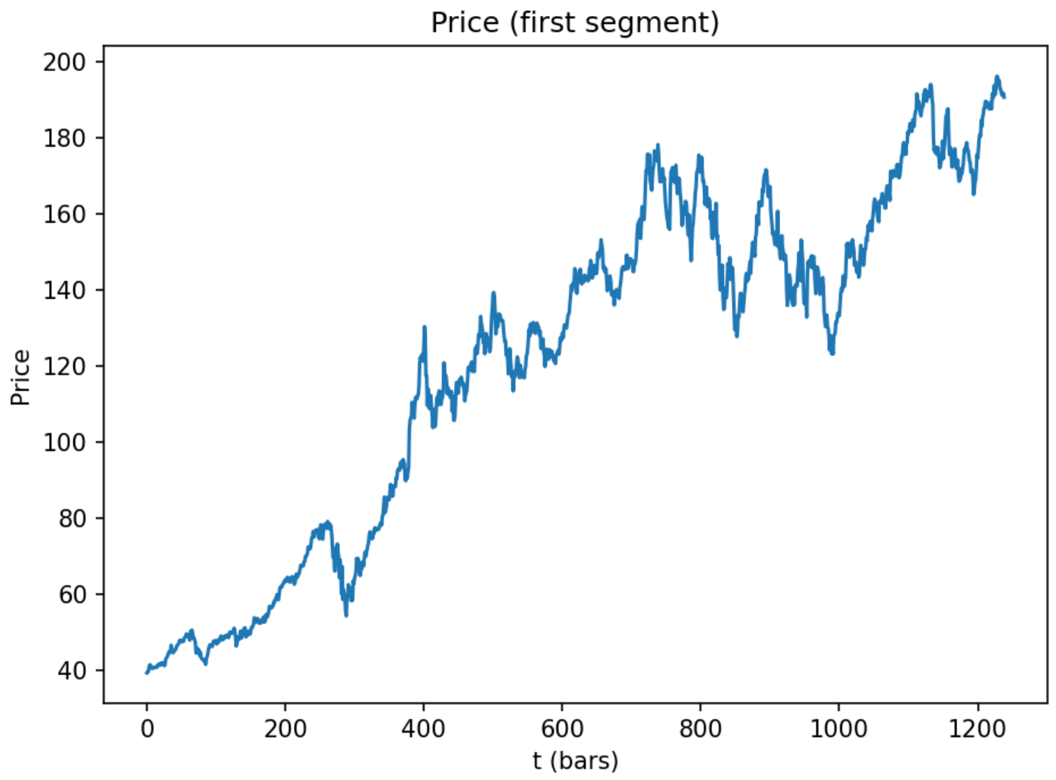


*Fig7****:****Underlying* ***market context*** *for the first ticker*

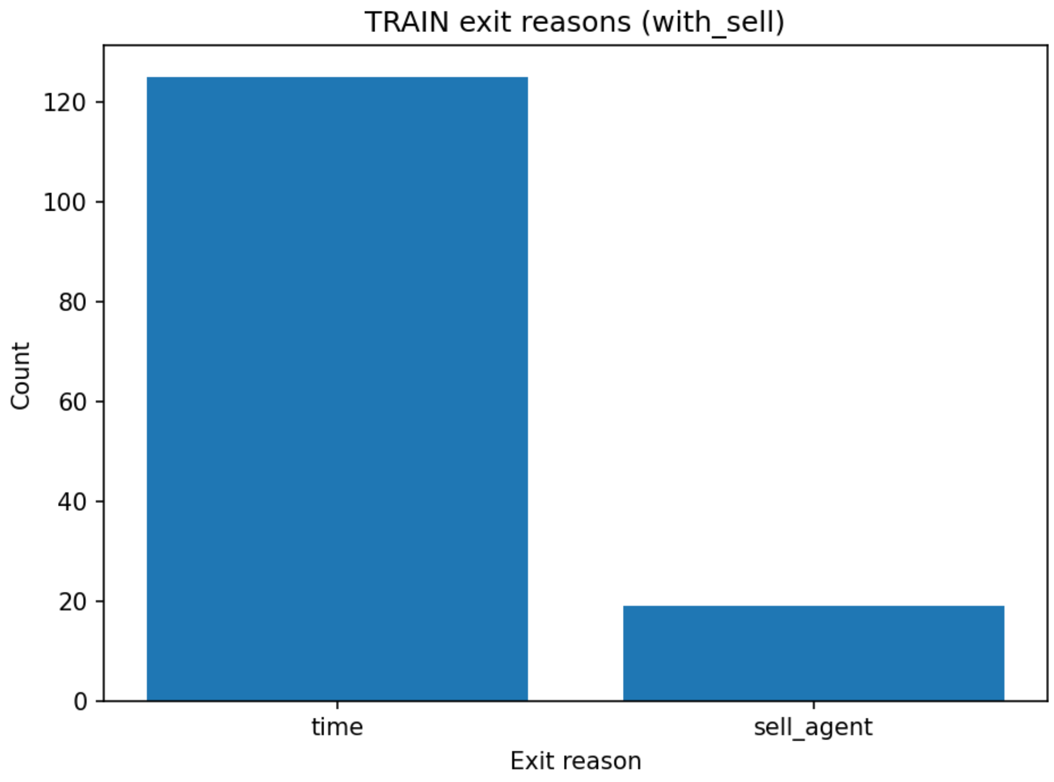


Figure 8: Exit reasons on the ***train set*** *with the Sell Agent enabled*

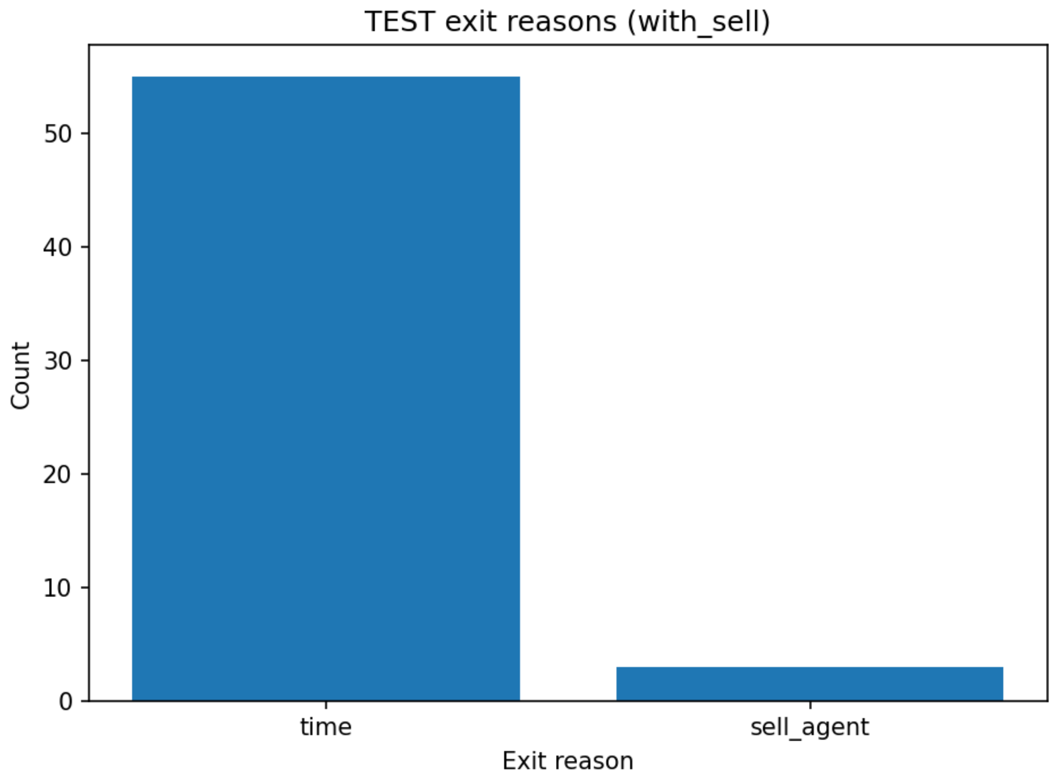


Figure 9: Exit reasons on the ***test set*** *with the Sell Agent enabled*

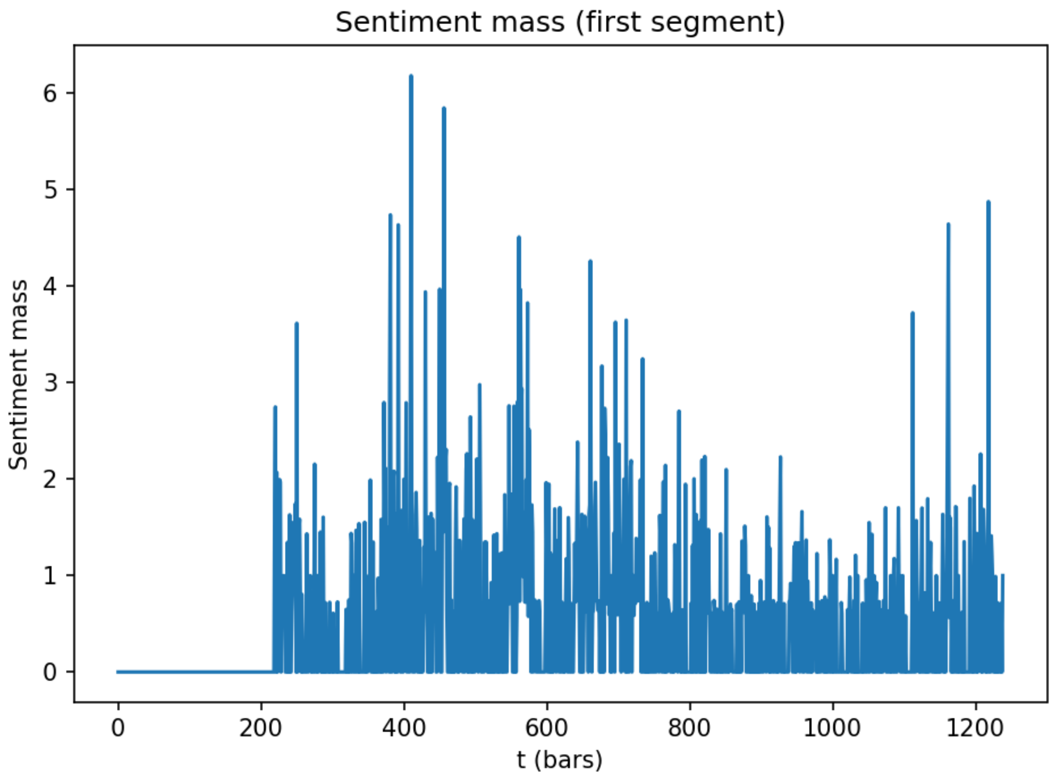


Figure 10: ***Sent. mass*** *reflects the vol. and relevance of news contributing to the daily sent. score*

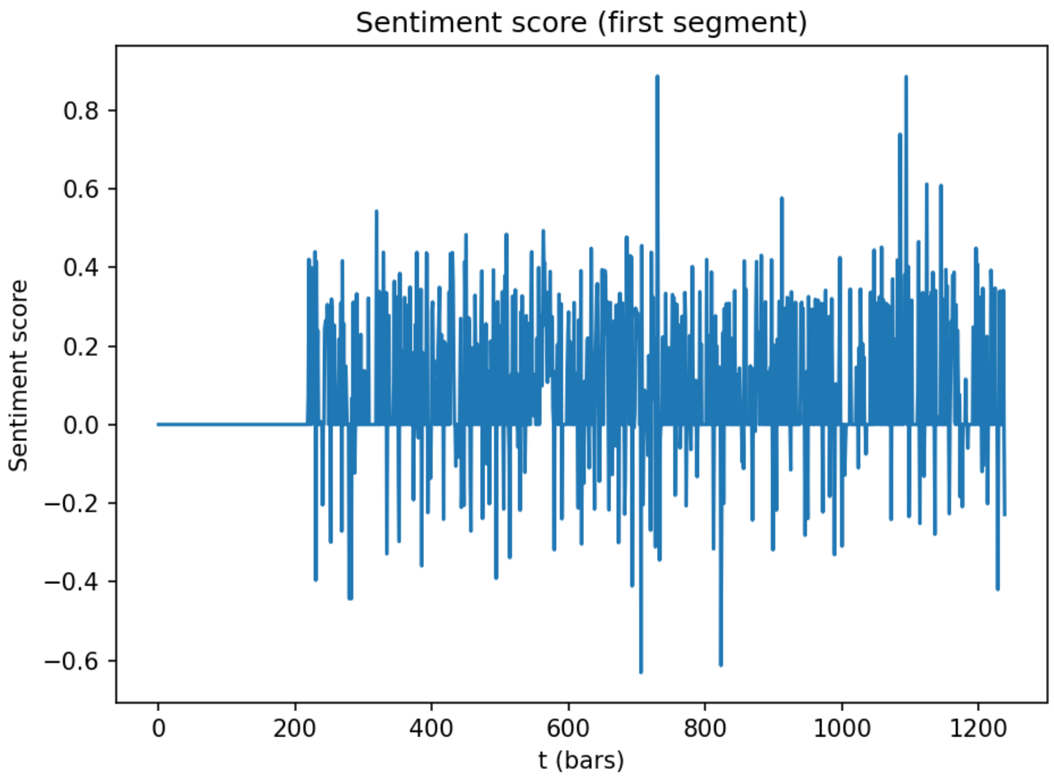


Figure 11: ***Sentiment polarity score*** *aligned with the same price segment.*

5.6.1 Synthesis of Sell Agent Evaluation

The plots provide coherent evidence that the Sell Agent and Trade Manager integration improves system-level behaviour without destabilising the trading pipeline. The sentiment mass and sentiment score visualisations confirm that sentiment information is event-driven, justifying its role as a contextual validation signal rather than a primary decision driver.
Exit-reason distributions show that the Sell Agent acts selectively: most positions still close at the time horizon, while only a limited number of agent-driven exits occur, particularly in the test set. This behaviour is expected at this stage of development and suggests that the confidence and gating thresholds are conservative rather than overly aggressive. The trade-level return distributions indicate that these selective early exits contribute to improved outcome distributions, particularly by reducing adverse trades rather than maximising extreme gains.

Finally, cumulative equity curves indicate consistent relative improvement over the Buy-only baseline in both train and test splits.

Overall, the Sell Agent functions as a stabilising enhancement to the Buy Agent rather than an override mechanism, validating the modular and confidence-based design of the proposed architecture.

Table 3

Dataset	Configuration	Trades	Final Equity	Avg Net Return	Win Rate	Sell Exits
Train	Buy-only	139	13.47	0.02356	0.619	0
Train	With Sell Agent	144	27.21	0.0274	0.639	19
Test	Buy-only	58	1.58	0.0134	0.603	0
Test	With Sell Agent	58	1.96	0.0182	0.603	3

5.6.2 Genetic Algorithm results

GA experiments further support that decision thresholds materially affect outcomes: by optimising parameters such as buy_min_confidence, sell_min_margin, sell_min_delta_vs_hold, min_hold_bars, and cooldown_steps, the system achieves higher final equity on the evaluation pass (using fixed trained agents). GA logs provide full traceability across generations and candidates.

TABLE 4: GENETIC ALGORITHM CONFIGURATION

Component	Setting
Population size	16
Generations	10
Random seed	42
Evaluation mode	Fixed trained Buy & Sell agents
Fitness metric	Validation-set final equity
Optimisation scope	Decision thresholds & trade constraint

After convergence, the best genome is evaluated once on the held-out test split to report final performance metrics.

TABLE 5: OPTIMIZED PARAMETERS (GENES)

Parameter	Description	Range
buy_min_confidence	Minimum Buy confidence threshold	[0.40, 0.75]
sell_min_margin	Minimum absolute margin to allow Sell	[0.00, 0.08]
sell_min_delta_vs_hold	Required improvement vs hold-to-horizon	[0.00, 0.05]
min_hold_bars	Minimum holding period before Sell	[0, 10]
cooldown_steps	Cooldown between consecutive trades	[0, 10]

TABLE 6: GA CONVERGENCE SUMMARY (TEST SET)

Generation	Best fitness (Final Equity)	Avg Net Return	Win Rate	Trades
0	3.60	0.0249	0.631	65
2	3.82	0.0244	0.647	68
4	3.94	0.0250	0.647	68
6	3.96	0.0262	0.656	64

Generation	Best fitness (Final Equity)	Avg Net Return	Win Rate	Trades
9 (final)	**4.17**	**0.0266**	**0.677**	**65**

TABLE 7: FINAL SELECTED CONFIGURATION

Parameter	Value
buy_min_confidence	0.485
sell_min_margin	0.0024
sell_min_delta_vs_hold	0.0054
min_hold_bars	2
cooldown_steps	1

The Genetic Algorithm shows consistent convergence toward configurations that improve out-of-sample final equity while preserving trade count and win rate. Fitness increases monotonically across generations, indicating a stable optimisation process rather than random fluctuation. This behaviour suggests that the fitness landscape, although noisy, is sufficiently structured to allow evolutionary search to identify stable improvements rather than random fluctuations.

It also confirms that decision thresholds impact system performance and that evolutionary optimisation provides a reliable alternative to manual tuning, specially in noisy financial environments.

5.7 Limitations and Evaluation Scope

The evaluation presented in this preliminary report is intended to validate system architecture, learning behaviour, and relative performance improvements rather than focusing only on establishing definitive trading profitability. All results are derived from historical backtests and should be interpreted accordingly.

At this stage of development, the system is not validated through traditional unit testing: my primary target is on verifying end-to-end behavioural correctness and learning stability rather than isolated functional components. As studied in the course, Reinforcement-learning systems are inherently stochastic and are more effectively validated through deterministic pipeline execution, fixed random seeds, and structured logging of decisions, rewards, and trade outcomes. So I decided to develop specific scripts to validate data integrity, reward computation, and execution logic under identical conditions. Once the system architecture is stable, I'll try to introduce formal unit tests for individual modules.

Also, at this stage I applied the Genetic Algorithm only to evaluation-time parameter optimisation, while training-time hyperparameter optimisation is reserved for future work. Also the sentiment data is incorporated as a filtering signal rather than a fully learned feature, prioritising interpretability and risk control over complexity.

User-facing components, including the NLP interface and web application, are not yet evaluated for this preliminary report, as their behaviour depends on the stability of the underlying trading engine. This phased development approach ensures that user interaction is built on top of a well-tested decision core rather than influencing learning behaviour prematurely.

Despite these limitations, the current evaluation provides strong evidence that the proposed architecture is functional, extensible, and suitable for further development toward a production-ready decision-support system.

Additional screenshots of the web interface and NLP integration layer are provided in Appendix A. The web interface serves primarily as an interaction layer exposing the validated trading engine and does not affect core decision logic.

5.8 Market Benchmark Comparison (Buy-and-Hold Baseline)

To contextualise the strategy’s performance, I compared the test-set results against passive buy-and-hold benchmarks for selected assets (AAPL, GOOGL, NVDA) and the S&P 500. Equity curves were normalised to 1.0 at the beginning of the test window to enable direct comparison of relative growth over the same evaluation period.

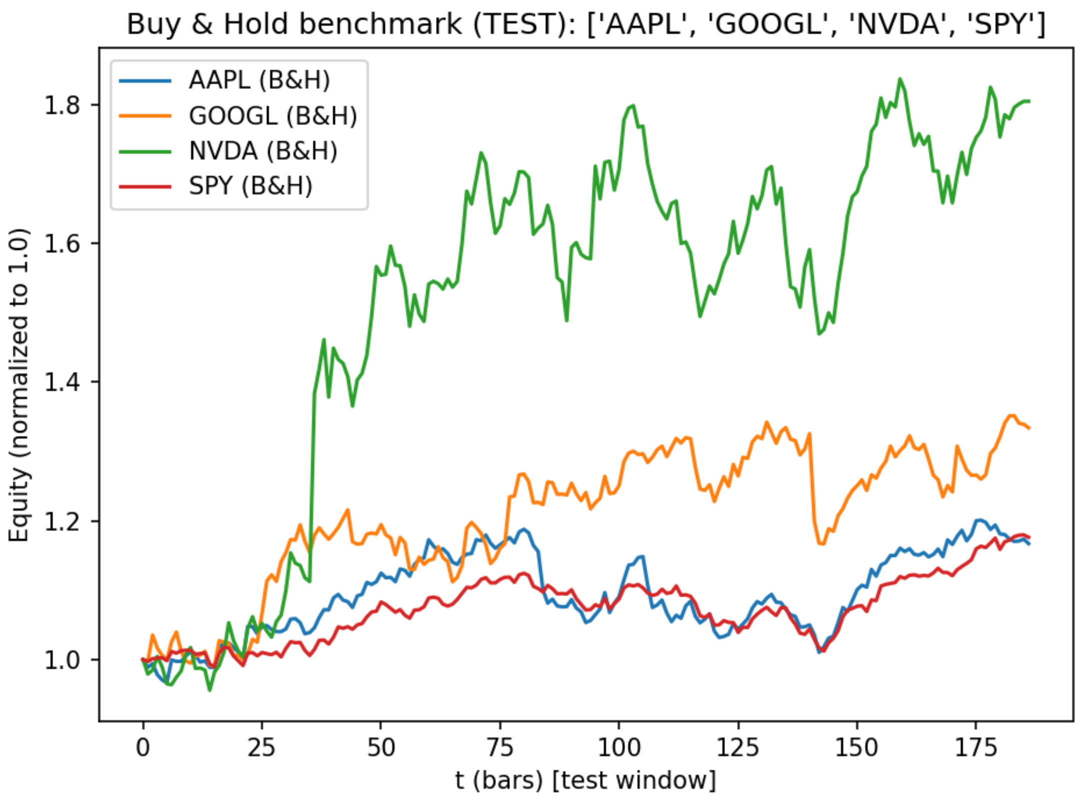


Figure 12: SPY vs tickers comparison

During the test window, buy-and-hold performance ranged between approximately 1.17× (SPY) and 1.80× (NVDA), reflecting a generally positive but mixed market regime. In comparison, the optimised trading strategy achieved a final test equity of 9.64, executing 129 trades with a win rate of 59.7%.

This indicates that the system's performance can not be simply attributed to broad market drift, but it also reflects selective exploitation of local price dynamics across assets.

This benchmark experiment uses the final GA-selected configuration and evaluates cumulative portfolio growth against a passive S&P 500 investment over the same period. Unlike the optimisation experiments in previous sections, this evaluation reflects the final deployed trading strategy rather than intermediate GA fitness runs.
It is also important to note that these results are derived from a single historical window and should not be interpreted as evidence of future profitability without further multi-regime validation.

6 Conclusion

Word Count: 301

My goal for this project was to design and develop a Financial Advisor Bot based on reinforcement learning, genetic optimisation, and explainable decision-making. I selected this project template to apply, extend, and critically evaluate the core concepts studied during the Artificial Intelligence course within a complex and realistic domain.
Many of the ideas I did explore in this project are based on reinforcement-learning examples studied during the course, such as Deep Q-Network agents trained to play Atari games in simulated environments. **I adopted the same underlying learning principles but applied them to a far more challenging and noisy environment: financial markets**. I intentionally designed the project with an ambitious scope. One of the key original aspects of my approach lies in how I adapted and extended standard reinforcement-learning concepts to better suit financial decision-making. I chose to pursue a multi-agent reinforcement-learning architecture with genetic optimisation and explainability to raise the technical bar and build a system that resembles a realistic financial decision-support tool rather than a simplified academic exercise.

I am aware that this ambition introduced significant challenges. Through this project, I observed that financial reinforcement learning is not only computationally expensive, but also highly sensitive to design choices and particularly prone to overfitting when evaluated solely on historical data.

The evaluation results show that the system I implemented can learn non-trivial trading behaviour and that Genetic Algorithms have a measurable impact on decision quality. A key objective of this project was to build a system grounded in a sound and flexible architecture, with stable learning behaviour, reproducible results, and the ability to support iterative improvement. From this perspective, the evaluation confirms the validity of the proposed design and provides a solid foundation for future extensions toward more robust evaluation and deployment.

References

[1] Salman Bahoo, Marco Cucculelli, Xhoana Goga, and Jasmine Mondolo. 2024. *Artificial intelligence in finance: A comprehensive review through bibliometric and content analysis.* SN Business & Economics 4. https://doi.org/10.1007/s43546-023-00618-x

[2] Francesco Bertoluzzo and Marco Corazza. 2012. *Testing different reinforcement learning configurations for financial trading: Introduction and applications.* Procedia Economics and Finance 3, 68–77. https://doi.org/10.1016/S2212-5671(12)00122-0

[3] Andrew G. Barto and Richard S. Sutton. 1996. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press.

[4] Charles Gold. 2003. *FX trading via recurrent reinforcement learning*. Proceedings of the IEEE International Conference on Computational Intelligence for Financial Engineering, 363–370.

[5] John Moody, Lizhong Wu, Yuansong Liao, and Matthew Saffell. 1998. *Performance functions and reinforcement learning for trading systems and portfolios.* Journal of Forecasting 17, 441–470.

[6] William D. Smart and Leslie Pack Kaelbling. 2000. *Practical reinforcement learning in continuous spaces*. Proceedings of the 17th International Conference on Machine Learning, 3–10.

[7] John H. Holland. 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press.

[8] John Moody and Matthew Saffell. 2001. *Learning to trade via direct reinforcement*. IEEE Transactions on Neural Networks 12, 4, 875–889. https://doi.org/10.1109/72.935097

[9] Bin Li and Steven C.H. Hoi. 2012. *Online portfolio selection: A survey*. ACM Computing Surveys 46. https://doi.org/10.1145/2512962

[10] PwC. 2017. *Sizing the prize: What's the real value of AI for your business and how can you capitalise?* PwC Report. https://www.pwc.com/gx/en/issues/analytics/assets/pwc-ai-analysis-sizing-the-prize-report.pdf

[11] Xinbo Ding, Shanwei Wu, and Zhiquan Luo. 2019. *Hybrid deep reinforcement learning with genetic algorithm for stock trading.* IEEE Access 7. https://doi.org/10.1109/ACCESS.2019.2957005

[12] Dong Zeng, Zhiqiang Wu, and Ming Li. 2023. *A hyperparameter adaptive genetic algorithm based on deep Q-learning network*. Journal of Circuits, Systems, and Computers. https://doi.org/10.1142/S0218126623502258

[13] Cong Ma, Jiangshe Zhang, Junmin Liu, Lizhen Ji, and Fei Gao. 2021. *A parallel multi-module deep reinforcement learning algorithm for stock trading*. Neurocomputing. https://www.sciencedirect.com/science/article/pii/S0925231221005233

[14] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. 2017. *A deep reinforcement learning framework for the financial portfolio management problem.* arXiv:1706.10059. https://arxiv.org/abs/1706.10059

[15] Da Woon Jeong and Yeong Hyeon Gu. 2024. *Pro Trader RL: Reinforcement learning framework for generating trading knowledge by mimicking the decision-making patterns of professional traders.* Expert Systems with Applications 124465. https://doi.org/10.1016/j.eswa.2024.124465

[16] Xiao-Yang Liu, Hongyang Yang, Qian Chen, Runjia Zhang, Liuqing Yang, Bowen Xiao, and Christina Dan Wang. 2020. *FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance*. arXiv:2011.09607. https://arxiv.org/abs/2011.09607

[17] AI4Finance-Foundation. *FinRL GitHub repository.* https://github.com/AI4Finance-Foundation/FinRL

Bibliography

Gregory Zuckerman. 2019. *The Man Who Solved the Market: How Jim Simons Launched the Quant Revolution.* Penguin Press.

Development

yFinance: https://github.com/ranaroussi/yfinance

AlphaVantage for news and sentiment API: https://www.alphavantage.co/documentation/

```
fastparquet
pyarrow
fastapi
pydantic
uvicorn
```