Software Engineering Department
Braude College of Engineering

Capstone Project - Phase A (61998)

Cryptopus

Reinforcement Learning-Based Meta-Strategy for Algorithmic Bitcoin Trading

Project Code: 26-1-D-29

By:
Omer Lev
Matan Reuven Tal

Advisor:
Dr. Reuven Cohen

February 2026

Project GitHub:
https://github.com/AlgoTrading-Capstone

Abstract

Algorithmic trading in cryptocurrency markets has gained significant popularity due to continuous data availability, high liquidity, and rapid price dynamics. Bitcoin, as the most established cryptocurrency, is a prominent candidate for automated trading systems. However, most existing approaches rely on a single strategy or a fixed set of indicators, which tend to perform well only under specific market conditions. As market regimes shift frequently, such static methods often degrade in performance and fail to adapt in real time.

This project proposes a reinforcement learning-based meta-strategy for algorithmic Bitcoin trading that dynamically combines and adapts multiple trading strategies according to prevailing market conditions. Each strategy is treated as a source of structured signals, which the reinforcement learning agent learns to integrate over time. The agent observes a comprehensive state representation that includes market data, technical indicators, portfolio and risk-related features, as well as encoded outputs of multiple strategies, and generates adaptive trading decisions accordingly.

The solution is implemented as a multi-component system consisting of four main parts. The first is a reinforcement learning training and evaluation project responsible for producing a trained trading agent using historical Bitcoin data. The second is a cloud-deployed server-side execution system that operates continuously, where the trained agent interacts with the exchange in real time. The third is a strategy conversion component that adapts successful open-source trading strategies into a standardized, system-compatible format. The fourth is a client-side application that enables monitoring, configuration and control of the platform. Together, these components form a coherent architecture that supports adaptive algorithmic trading.

By framing trading as a decision-making problem over multiple competing strategies, this project aims to demonstrate that reinforcement learning can effectively support adaptive strategy selection and risk-aware trading. The expected outcome is a flexible and extensible trading platform that improves upon static strategy-based approaches and provides a solid foundation for future research and development in intelligent financial trading systems.

Keywords: *Reinforcement Learning, Algorithmic Trading, Bitcoin, Multi-Strategy, Cloud Deployment*

Table of Contents

1. Introduction

1.1. Background

1.1.1. Algorithmic Trading in Financial Markets

Algorithmic trading refers to the use of computer programs and mathematical models to make trading decisions and execute orders in financial markets. Instead of relying on human judgment alone, algorithmic systems analyze market data, apply predefined rules or learned policies and react automatically to market conditions. Over the past two decades, algorithmic trading has become a dominant approach in modern financial markets, driven by advances in computing power, data availability and automation technologies.

According to market research by Mordor Intelligence, the global algorithmic trading market is estimated at approximately USD 18.7 billion in 2025 and is projected to reach USD 28.4 billion by 2030, reflecting the widespread adoption of automated trading systems across financial markets. [1]

The main motivation behind algorithmic trading is the ability to process large volumes of market information quickly and consistently. Algorithms can monitor prices, volumes, and technical indicators in real time, identify patterns that are difficult for humans to detect, and execute trades with high speed and precision. Unlike human traders, algorithms are not affected by emotions such as fear, greed, or fatigue, which often lead to inconsistent or suboptimal decisions.

Another key advantage of algorithmic trading is the ability to test trading ideas on historical data before deploying them in real markets. Through backtesting and simulation, strategies can be evaluated, compared and refined in a controlled environment. This reduces reliance on intuition and allows decisions to be based on measurable performance metrics such as returns, risk and drawdowns. Overall, algorithmic trading aims to improve efficiency and consistency compared to traditional manual trading.

1.1.2. Bitcoin and Cryptocurrency Markets

Bitcoin is the first and most widely adopted cryptocurrency, operating on a decentralized blockchain network. Since its introduction, Bitcoin has evolved from a niche technological experiment into a globally traded digital asset with significant market capitalization and liquidity. Today, Bitcoin is actively traded on numerous exchanges and serves as the primary reference asset for the broader cryptocurrency market.

Unlike most traditional financial markets, cryptocurrency markets operate continuously, 24 hours a day and 7 days a week, without weekends or holidays. In addition, market data such as prices, volumes, and order information is openly accessible through exchange APIs[1], enabling extensive historical and real-time analysis in a transparent, data-rich trading environment.

[1] **Exchange API:** An application programming interface provided by a cryptocurrency exchange that allows external systems to programmatically access market data and submit trading orders.

1.1.3. Why Bitcoin Is Well-Suited for Algorithmic Trading

Its continuous 24/7 trading nature allows algorithms to operate without interruption, while high market liquidity ensures frequent price movements and trading opportunities.

Unlike traditional assets such as stocks, Bitcoin does not rely on fundamental indicators such as earnings reports, balance sheets or dividends. Instead, its price is primarily driven by supply and demand dynamics, market sentiment and trader behavior, making it particularly suitable for quantitative and data-driven approaches focused on price action, volatility, and technical patterns.

Furthermore, Bitcoin's relatively high volatility compared to traditional assets increases both risk and potential reward, creating frequent trading opportunities for algorithmic systems. As a result, Bitcoin has become a popular target for automated trading systems, ranging from simple rule-based strategies to advanced machine learning models.

1.2. Problem Statement

1.2.1. Strategy Sensitivity to Market Conditions

Bitcoin trading relies on a wide variety of strategies based on different signal types, such as technical indicators, market sentiment, derivatives data, and news events (see **Appendix A** for an illustrative example of a trading strategy). While each of these strategies can be effective, their performance is typically limited to specific market conditions. For example, a trend-following strategy may perform well during trending markets but fail during sideways or highly volatile periods. As market behavior shifts over time, strategies that previously generated consistent returns often become less effective, as shown in empirical studies of cryptocurrency markets. [2]

1.2.2. Conflicting Signals in Multi-Strategy Trading Systems

To address the sensitivity of individual strategies to shifting market conditions, traders often attempt to use multiple strategies simultaneously. However, combining several strategies introduces a new challenge: conflicting signals. It is common for one strategy to indicate a buy signal while another suggests selling or holding a position. Without a clear and systematic method to resolve these conflicts, decision-making becomes complex and inconsistent. [3] Manual interpretation of multiple signals does not scale well and increases the risk of human error, especially in fast-moving markets such as Bitcoin.

1.2.3. Limited Accessibility of Advanced Algorithmic Trading Infrastructure

Advanced algorithmic trading systems typically require substantial computational resources, sophisticated algorithms, and access to high-quality data. These capabilities are usually available to institutional players such as hedge funds and financial institutions, but are far less accessible to retail traders. As a result, there exists a significant gap between institutional and individual participants in terms of technological capabilities and trading performance. [4]

1.2.4. Knowledge Barriers in Algorithmic Trading Adoption

Beyond infrastructure and resources, algorithmic trading also presents notable knowledge barriers. Developing and operating automated trading systems requires a solid background in programming, data analysis and software engineering. At the same time, effective trading demands domain knowledge in financial markets, including risk management, market mechanics and trading instruments. The need to combine both technical and financial expertise makes advanced algorithmic trading solutions difficult to adopt for many potential users. [5]

Together, these challenges expose the limitations of traditional rule-based trading approaches and the difficulty of managing multiple strategies in dynamic market conditions. This motivates the need for a system that can systematically combine diverse strategies, adapt to changing market behavior, and reduce reliance on manual decision-making, while remaining accessible and practical to implement.

1.3. Existing Solutions and Limitations

Following the identification of key challenges in algorithmic Bitcoin trading, this section reviews existing solutions that are commonly used in practice. These solutions provide varying levels of automation and accessibility, however, as shown below, none of them fully address the core limitations outlined in the Problem Statement.

1.3.1. Exchange-Native Trading Bots

Exchange-native trading bots are automated tools integrated directly into cryptocurrency exchanges and primarily target retail users. These bots typically require no external infrastructure, programming knowledge, or direct interaction with exchange APIs and are configured through graphical interfaces and predefined templates.

Common examples include the built-in trading bots offered by major exchanges such as Binance, as well as exchanges like Pionex that focus on integrated automated trading tools. The primary advantage of this approach is its high accessibility and ease of use.

1.3.2. Third-Party Automated Trading Platforms

Third-party automated trading platforms are external Software-as-a-Service (SaaS) solutions that connect to cryptocurrency exchanges via APIs. They typically offer rule-based trading bots or signal-based automation, targeting retail users who seek more advanced automation without building custom systems.

Representative examples include platforms such as 3Commas and Coinrule. These solutions reduce development and operational overhead by allowing users to deploy automated strategies without managing infrastructure or writing complex code.

1.3.3. Script-Based Trading Systems

Script-based trading systems allow users to define custom trading logic through code, offering full control over strategy behavior. A prominent example is TradingView, which enables strategy development using its proprietary Pine Script language.

This approach is popular among technically inclined traders because it supports historical backtesting, flexible strategy design and widespread sharing of scripts within the trading community. However, this flexibility comes at the cost of significant complexity. Users must possess both programming skills and solid knowledge of trading strategies to design, test, and maintain effective strategies.

1.3.4. Common Limitations of Existing Approaches

Despite their differences in accessibility and flexibility, the solutions described above share several fundamental limitations. Most rely on fixed or manually adjusted strategies that are sensitive to specific market conditions and do not adapt automatically as market behavior changes.

Moreover, none of these approaches provide a unified meta-decision layer capable of systematically integrating multiple strategies and resolving conflicting signals in real time. As a result, effective operation often depends on continuous user intervention and substantial technical and financial expertise, limiting scalability and robustness in dynamic market environments.

2. Literature Review

2.1. Introduction to the Literature Review

As discussed in the Problem Statement, trading strategies in Bitcoin markets are highly sensitive to changing market conditions, and no single strategy performs consistently across different regimes. Combining multiple strategies introduces conflicting signals, increasing decision complexity and exposing the limitations of manual and rule-based approaches in dynamic market environments.

These challenges point to the need for a decision-making mechanism that can operate under non-stationary conditions, integrate multiple sources of information and continuously adjust its behavior based on observed outcomes. Rather than relying on fixed rules or static strategy selection, such a mechanism must be capable of learning from interaction with the market environment and updating its decisions over time. Within the existing literature, reinforcement learning emerges as a natural framework for addressing these requirements, as it formulates trading as a sequential decision-making problem and enables adaptive policy learning in dynamic and uncertain environments. [6]

This suitability is further illustrated by empirical results reported by Liu et al. [13], who compare a reinforcement learning-based trading strategy (using DDPG RL algorithm) with traditional baselines such as Buy-and-Hold and a minimum-variance[2] portfolio. As shown in Figure 1, the RL-based approach achieves higher cumulative portfolio value over an extended time horizon when compared to the Dow Jones Industrial Average (DJIA) and a minimum-variance portfolio, resulting in excess returns and highlighting its ability to adapt to changing market conditions.

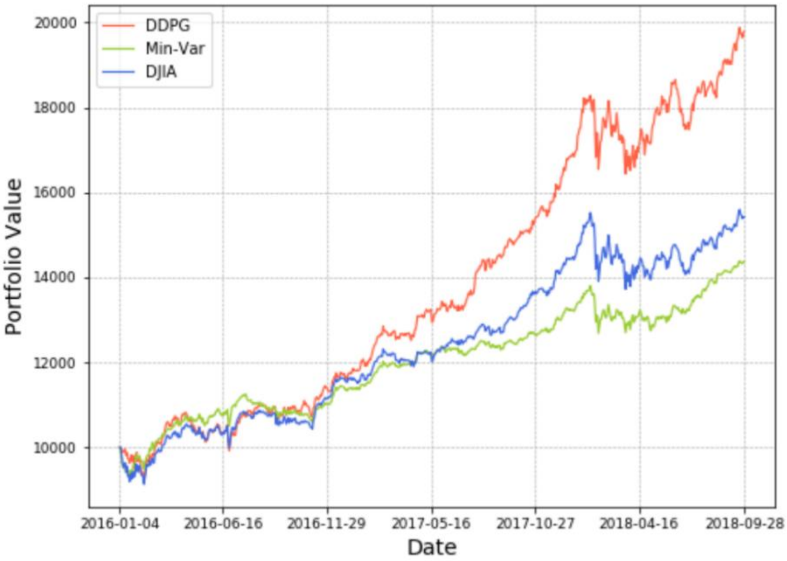

Figure 1: Performance comparison of a reinforcement learning-based trading strategy (DDPG), Buy-and-Hold and minimum-variance baselines. Source: Liu et al. [13]

Accordingly, the following sections review relevant theoretical and practical work on reinforcement learning in financial markets, with a focus on its suitability for strategy integration, conflict resolution, and operation in continuously changing market conditions.

[2] **Minimum-variance:** refers to a portfolio construction approach from modern portfolio theory that aims to minimize overall portfolio risk (variance of returns), without explicitly targeting return maximization. It is commonly used as a low-risk benchmark in portfolio performance comparisons.

2.2. Reinforcement Learning Fundamentals

2.2.1. Reinforcement Learning Overview

Reinforcement learning (RL) is a machine learning paradigm concerned with learning how to act in an environment through trial and error. In RL, an agent interacts with its environment by observing the current state, selecting an action, and receiving feedback in the form of a reward. As a result of its action, the environment transitions to a next state, making the decision process inherently sequential and closed loop.

The objective of the agent is to learn a policy- a mapping from states to actions that maximizes the expected cumulative reward over time. Learning is driven by experience gathered through interaction, allowing the agent to improve its behavior based on evaluative feedback that indicates how good an action was, rather than explicit instructions about which action should be taken.

A fundamental challenge in this process is balancing exploration, where the agent tries new actions to acquire information, and exploitation, where it leverages existing knowledge to maximize reward. Through repeated interaction, the agent gradually refines its policy and adapts its decisions according to observed outcomes. [7]

2.2.2. Markov Decision Process Formulation

A standard and widely used formalization of reinforcement learning problems is the Markov Decision Process (MDP). An MDP is defined as a tuple:

$$M = \langle S, A, P, R, \gamma \rangle$$

where:

- S is the set of possible states
- A is the set of possible actions
- $P(s_{t+1} \mid s_t, a_t)$ is the state transition probability function
- $R(s_t, a_t)$ is the reward function
- $\gamma \in [0,1]$ is the discount factor

At each discrete time step t, the agent observes the current state $s_t \in S$, selects an action $a_t \in A$, receives a reward r_{t+1}, and the environment transitions to a new state s_{t+1} according to the transition dynamics P.

The indexing r_{t+1} reflects that both the reward and the next state are determined as a consequence of the action taken at time t.

The Markov property assumes that the next state and reward depend only on the current state and action, and not on the full history of previous states and actions. While this assumption may not hold exactly in real-world settings, it is commonly approximated by designing the state representation to include sufficient information for effective decision-making. [7]

2.2.3. Objective and Long-Term Reward Optimization

The objective of reinforcement learning is not to maximize immediate reward at each time step, but to maximize the cumulative reward obtained over time. This objective is formalized through the discounted return, defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where R_{t+k+1} denotes the reward received after future actions, and $\gamma \in [0,1]$ is the discount factor. The discount factor controls the trade-off between short-term and long-term outcomes: values of γ close to 1 place greater emphasis on future rewards, while lower values prioritize more immediate gains. [7]

2.2.4. Policies, Value Functions and Actor-Critic Methods

RL methods can be broadly categorized into value-based and policy-based approaches. Value-based methods aim to learn a value function that estimates the expected future reward associated with a state or a state-action pair. Policy-based methods, in contrast, learn the policy directly, typically by optimizing it using gradient-based techniques.

Many modern RL algorithms combine these ideas through an actor-critic architecture. In this framework, the actor represents the policy that selects actions, while the critic estimates a value function that evaluates the quality of the actor's decisions. This separation often leads to improved stability and learning efficiency, especially in complex environments. [7]

2.2.5. Exploration and Problem Formulation

A fundamental challenge in reinforcement learning is the exploration-exploitation trade-off. To improve its policy, an agent must explore actions that may yield better outcomes, while also exploiting actions that are already known to perform well. Effective RL systems balance these two requirements using mechanisms such as stochastic policies[3] or entropy regularization[4]. [7]

Finally, the performance of an RL system depends strongly on how the problem is formulated. The definition of the state representation, action space, reward function, and episode structure has a critical impact on learning behavior. In many applied settings, careful environment and reward design are as important as the choice of learning algorithm itself.

[3] **Stochastic Policy:** A decision-making mechanism that maps states to a probability distribution over possible actions, rather than selecting a single fixed action. This approach ensures continuous exploration of the environment by assigning a non-zero probability to multiple actions.

[4] **Entropy Regularization:** An optimization technique that augments the objective function with an entropy term. This encourages the agent to maintain a higher degree of randomness in its policy, thereby promoting continuous exploration and preventing premature convergence to suboptimal deterministic behaviors.

2.3. RL for Decision-Making in Financial and Cryptocurrency Markets

Financial markets present a challenging environment for automated decision-making due to their dynamic and non-stationary nature, where changing macroeconomic conditions, participant behavior, liquidity and external events cause market regimes to evolve over time. In such settings, assumptions underlying fixed decision rules often break down, leading to performance degradation.

Reinforcement learning is well suited to this environment because it models trading as a sequential decision-making process under uncertainty and optimizes long-term outcomes through continuous policy updates based on observed rewards and state transitions. By learning through ongoing interaction, RL can adapt its behavior as market conditions change rather than relying on static assumptions. [6]

Another key challenge in trading systems is the integration of multiple information sources and strategies, which may provide partial or conflicting signals. Within the RL framework, such information can be incorporated into the state representation, enabling the agent to learn a unified decision policy that resolves these conflicts based on long-term performance.

These challenges are further amplified in cryptocurrency markets, particularly Bitcoin, which operate continuously, exhibit high volatility and undergo rapid regime shifts. At the same time, the availability of high-frequency market data makes these markets well suited for data-driven learning approaches, and prior studies demonstrate the potential of reinforcement learning to operate effectively under such conditions. [8]

Despite this potential, practical challenges related to reward design, learning stability, and realistic evaluation remain, motivating careful algorithm selection and system design, as discussed in the following sections.

2.4. RL Algorithm Selection for Continuous Decision-Making

Selecting an appropriate reinforcement learning algorithm is a critical step when applying RL to financial and cryptocurrency trading. Trading decisions are inherently sequential and often continuous, requiring algorithms that operate reliably under noisy and non-stationary conditions.

One important consideration is the structure of the action space. In trading systems, actions are rarely limited to simple discrete choices. Instead, decisions often involve continuous quantities such as position sizing, portfolio allocation, or exposure levels. Algorithms that naturally support continuous action spaces are therefore more suitable for such tasks than methods that rely on discretization, which restrict decisions to a small set of fixed actions. Continuous control further enables more granular risk management, allowing the agent to adjust exposure incrementally rather than being constrained to coarse, binary trading decisions. [6]

Another key factor is learning stability. Financial reward signals are noisy and sparse, and small changes in policy can lead to large variations in observed returns. Algorithms that exhibit unstable learning dynamics may perform well during training but fail to generalize or remain robust when market conditions change. As a result, methods that emphasize stable policy updates and controlled exploration are generally preferred in trading applications. [8]

From a methodological perspective, these requirements tend to favor policy-based and actor-critic approaches over purely value-based methods. While value-based algorithms have been successfully applied in many domains, they often struggle with continuous control, sensitivity to function approximation errors, and instability in highly stochastic environments. In contrast, policy-based methods optimize the decision policy directly and are better suited to modeling continuous and probabilistic actions.

2.4.1. Policy-Based and Actor-Critic Methods

Policy-based reinforcement learning methods directly learn a parameterized policy that maps states to actions, making them well suited for continuous and stochastic decision problems such as financial markets. Actor-critic methods extend this approach by combining a policy (the actor) with a value estimator (the critic), which evaluates the expected future return of the actor's decisions. This separation allows the critic to provide structured feedback that reduces variance in policy updates, leading to improved learning stability and efficiency.

In the context of trading, actor-critic architectures are particularly attractive because they balance flexibility and control. The actor can represent complex trading behaviors, while the critic helps guide learning in environments characterized by delayed rewards and high uncertainty. As a result, actor-critic methods are widely adopted in the reinforcement learning literature applied to financial and cryptocurrency markets. [8]

2.4.2. Proximal Policy Optimization (PPO) Algorithm

Proximal Policy Optimization (PPO) is an on-policy actor-critic algorithm designed to improve learning stability while maintaining implementation simplicity. Its central idea is to limit the magnitude of policy updates by constraining changes between successive policies. This is typically achieved through a clipped objective function that discourages overly large updates, reducing the risk of destabilizing the learning process.

These properties make PPO well suited for trading environments, where abrupt policy changes can lead to erratic behavior and significant losses. PPO has been shown to perform reliably across a wide range of tasks and is frequently used as a baseline in reinforcement learning research. Its balance between stability, performance, and ease of tuning has contributed to its popularity in financial RL applications. [8]

In trading contexts, PPO's on-policy nature encourages learning from recent market interactions, which can be beneficial in non-stationary environments. However, this comes at the cost of lower sample efficiency compared to some off-policy methods, motivating the consideration of complementary algorithms with higher sample efficiency.

2.4.3. Soft Actor-Critic (SAC) Algorithm

Soft Actor-Critic (SAC) is an off-policy actor-critic algorithm that extends the standard reinforcement learning objective by explicitly encouraging exploration through entropy maximization[5]. In addition to maximizing expected return, SAC seeks to maximize the entropy of the policy, promoting more diverse action selection and reducing premature convergence to suboptimal strategies. [9]

This exploration-driven objective is particularly advantageous in financial markets, where the reward landscape may change over time and previously successful strategies can lose effectiveness. By maintaining a higher degree of exploration, SAC can adapt more effectively to evolving market conditions. Furthermore, its off-policy nature allows it to reuse past experience more efficiently, improving sample efficiency and reducing the amount of data required for learning.

SAC has been successfully applied to a variety of continuous control problems and has gained increasing attention in financial reinforcement learning research. Its ability to balance exploration and exploitation in a principled manner makes it a strong candidate for trading systems that operate continuously and must remain adaptive under uncertainty. [10]

2.5. Reward Design and Evaluation in Trading RL

Reward design plays a central role in applying reinforcement learning to trading systems. In trading applications, the objective is typically framed as improving long-term performance while controlling risk, and the success of RL-based trading agents depends strongly on carefully designed reward formulations.

In practice, reward functions should reflect not only profitability but also risk and market frictions. Focusing only on raw profit (e.g., total return) can lead to unstable or excessively risky behavior, while more principled formulations integrate elements such as transaction costs and risk-sensitive performance considerations (e.g., risk-adjusted objectives or drawdown-aware penalties), rather than optimizing profit alone. [11]

Another challenge is that reward signals in trading can be delayed and noisy: the effect of an action may only become apparent after multiple time steps. [11] This connects directly to the "temporal credit assignment problem" in reinforcement learning- determining how to attribute a delayed outcome (reward) to the earlier sequence of actions that contributed to it. Consequently, reward functions should be specified carefully so that learning signals remain informative while avoiding unintended incentives. [7]

Evaluation methodology is equally critical. Backtesting on historical data is the standard approach for assessing trading strategies, but it carries inherent limitations. In particular, overly strong backtest results may reflect overfitting to specific historical periods and evaluation setups that do not faithfully

[5] **Entropy Maximization:** A reinforcement learning objective where the agent seeks to maximize both the expected reward and the entropy of its policy. This approach encourages the agent to maintain stochastic behavior, preventing premature convergence to suboptimal deterministic strategies and improving exploration in complex environments.

represent future conditions or realistic execution. The literature therefore emphasizes robust evaluation practices, including strict time-based train-test separation, realistic modeling of transaction costs, and performance assessment across diverse market conditions. [11]

Together, these considerations demonstrate that effective application of reinforcement learning to trading depends not only on algorithm selection but also on principled reward design and rigorous evaluation.

2.6. Alternative Learning Paradigms for Algorithmic Trading

Several learning paradigms have been applied to algorithmic trading. Traditional rule-based approaches discover trading opportunities using handcrafted heuristics, but often exhibit poor generalization and tend to perform well only under specific market conditions. A common data-driven alternative is prediction-based trading, where supervised learning[6] models are trained to forecast future prices, returns, or directional movements and trading decisions are derived from these predictions. However, the high volatility and noisy nature of financial markets make accurate forecasting difficult, and there is a well-recognized gap between prediction signals and profitable trading actions, limiting the effectiveness of prediction-first pipelines. In contrast, reinforcement learning enables training an end-to-end agent that maps observed market information directly to trading actions and optimizes long-term objectives within a unified framework. [11]

2.7. Summary and Identified Gap

The reviewed literature demonstrates that reinforcement learning provides a principled and flexible framework for addressing sequential decision-making problems in financial and cryptocurrency markets. Prior studies highlight the suitability of RL for operating under non-stationary conditions, handling delayed and noisy reward signals, and optimizing long-term performance rather than isolated predictions. Actor-critic algorithms, in particular, are widely adopted due to their stability and ability to support continuous decision space.

At the same time, existing research and implementations reveal several limitations. Many approaches focus on single-strategy optimization or assume fixed market conditions, offering limited mechanisms for systematically integrating multiple trading strategies or resolving conflicting signals. Moreover, practical challenges related to reward design, evaluation, and adaptability often constrain real-world applicability.

These gaps motivate the need for a reinforcement learning-based trading framework that acts as a meta-decision layer, enabling dynamic strategy integration and adaptation to changing market conditions. This motivation leads directly to the proposed solution in the following section.

[6] **Supervised Learning:** a machine learning paradigm in which models are trained on labeled historical data to learn a fixed mapping between input features and predefined target outputs, typically by minimizing a prediction error.

3. Proposed Solution

3.1. High-Level Overview

The proposed solution is a reinforcement learning-based algorithmic trading system for Bitcoin, designed as a meta-strategy rather than a single standalone trading strategy. At the core of the system is a reinforcement learning agent that serves as a centralized decision-making layer. The agent integrates the outputs of multiple heterogeneous trading strategies together with market state information and learns to dynamically combine and weight these inputs in order to produce coherent trading decisions. This approach directly addresses the challenge of conflicting signals and enables adaptation to changing market conditions over time.

To support continuous and stable decision-making in a noisy and non-stationary market environment, the system employs policy-based reinforcement learning algorithms suited for continuous control, specifically Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). These algorithms were selected due to their stability, robustness, and suitability for long-term sequential decision processes typical of financial trading.

The overall system is structured as three independent but interconnected sub-projects, allowing for modular development, clear separation of responsibilities and scalability. The first sub-project focuses on training and evaluation of reinforcement learning agents using historical Bitcoin market data. It includes a dedicated training and evaluation pipeline that produces trained models, performance metrics and backtesting results, which are later deployed in the live environment.

The second sub-project is a client-server system designed to operate continuously in a cloud environment. The server-side component runs 24/7 and is responsible for executing the trained reinforcement learning agent in real time to generate trading decisions. A desktop client application provides monitoring, management and visualization capabilities, supporting two user roles: a regular user and an administrator, each with different levels of access and control.

The third sub-project addresses the integration of trading strategies into the system. Since many widely used open-source trading strategies are implemented in TradingView's Pine Script language, this component introduces an automated strategy conversion process. It is based on an AI-driven agent that translates Pine Script strategies into a unified Python-based format compatible with the system, enabling efficient expansion of the strategy pool.

The following sections describe each of these sub-projects in detail, including their requirements, architecture, technologies and design considerations.

3.2. Sub-Project I: RL Training and Evaluation

Development status: advanced development stage. Initial trained reinforcement learning models have already been produced and evaluated.

3.2.1. Overview

The training and evaluation sub-project is a dedicated component responsible for training, evaluating, and backtesting reinforcement learning agents for Bitcoin trading using historical market data. This component is intentionally separated from the live trading and execution environment and focuses exclusively on model development, experimentation and performance assessment. It does not perform real-time trading and is not exposed to end user.

The system is operated through a developer-oriented command-line interface (CLI), which allows controlled execution of the training and evaluation workflows. The CLI supports multiple execution modes, including training combined with backtesting, training only, and backtesting of previously trained models. This design enables iterative development and systematic evaluation of reinforcement learning agents under different configurations and data ranges.

Training is performed on historical Bitcoin OHLCV[7] data obtained from cryptocurrency exchanges, enriched with multiple layers of market information. These include technical indicators, risk-related features and the outputs of multiple trading strategies.

The learning process is carried out within a custom trading simulation environment that models realistic trading behavior. The environment captures portfolio dynamics over time, including:

- Account balance and total portfolio equity
- Open positions, including position direction and size
- Transaction costs such as trading fees and slippage[8]
- Leverage[9] constraints that limit maximum exposure
- Position limits enforcing caps on holdings
- Stop-loss[10] mechanisms with dynamic placement and execution logic

The agent operates in a continuous action space, where actions control both market exposure and stop-loss behavior.

Model training is performed using policy-gradient reinforcement learning algorithms within a fully configurable training framework. The framework exposes configuration controls over learning

[7] **OHLCV:** refers to a standard market data representation that includes the Open, High, Low, and Close prices of an asset within a given time interval, along with the corresponding Trading Volume.

[8] **Slippage:** refers to the difference between the expected execution price of a trade and the actual price at which the trade is executed. It typically occurs due to market volatility or limited liquidity.

[9] **Leverage:** refers to the use of borrowed capital to increase market exposure relative to the trader's actual account balance.

[10] **Stop-loss:** refers to a predefined risk control mechanism that automatically closes an open position when the asset price reaches a specified unfavorable level.

algorithms, optimization hyperparameters, trading and environment constraints, and feature selection. These controls enable systematic experimentation and reproducible evaluation of trained agents under varying market conditions and modeling assumptions. A complete specification of the supported configuration parameters is provided in **Appendix B**.

The outputs of this sub-project include:

- Trained reinforcement learning model
- Metadata files describing the full training configuration and environment setup
- Backtesting results based on unseen historical data, including:
  - Step-level logs capturing the agent's state, actions and rewards over time
  - Trade-level logs detailing individual trades and position changes
  - Quantitative performance metrics summarizing returns and risk characteristics
  - Visual performance comparisons against baseline strategies such as buy-and-hold

These outputs are used by subsequent system components and provide the basis for selecting and deploying trained agents in the operational trading system.

3.2.2. Requirements

The RL Training and Evaluation sub-project is designed to satisfy a set of functional and non-functional requirements that ensure correctness, reproducibility, and reliable performance assessment of trained agents. These requirements define the supported training and evaluation workflows, configuration flexibility, data handling and output artifacts, as well as quality attributes such as modularity, scalability and experiment repeatability.

A complete and structured specification of the functional and non-functional requirements for this sub-project is provided in **Appendix C**.

3.2.3. Training and Evaluation Pipeline - System Flow

The process is initiated through a command-line interface, where the developer selects the execution mode and provides a start and end date for the experiment. Based on these dates, the system automatically applies a predefined train-test split to separate data used for training from data reserved for evaluation.

Throughout the entire pipeline, a centralized logging mechanism records execution events. This logging layer enables monitoring, supports debugging and post-run analysis.

In the data preparation phase, historical market data is loaded and consolidated into a unified dataset. A smart data retrieval mechanism ensures that only missing or outdated data segments are downloaded, avoiding redundant downloads across repeated executions. The data is then cleaned and processed, feature engineering is applied, and multiple trading strategies are executed to generate aligned strategy signals. These components are combined into structured state representations, which are later normalized within the trading environment to ensure stable and consistent learning behavior.

When training is enabled, the prepared data is used to initialize the trading simulation environment and configure the reinforcement learning algorithm and network architecture. Training is performed using the ElegantRL framework, which manages the training loop through iterative environment interaction and periodic policy evaluation. The agent alternates between collecting experience from the environment and updating its policy parameters, while evaluation runs are executed at fixed intervals to assess performance. This process continues until the configured training horizon is reached, after which the trained model and its metadata are saved. [12]

In backtesting mode, a trained model is loaded and validated against the current configuration. The model is executed in inference mode, while detailed execution logs are collected. Performance metrics are computed and visual artifacts are generated to summarize trading behavior and results.

The pipeline concludes by persisting all outputs, including trained models, logs, metrics and plots.

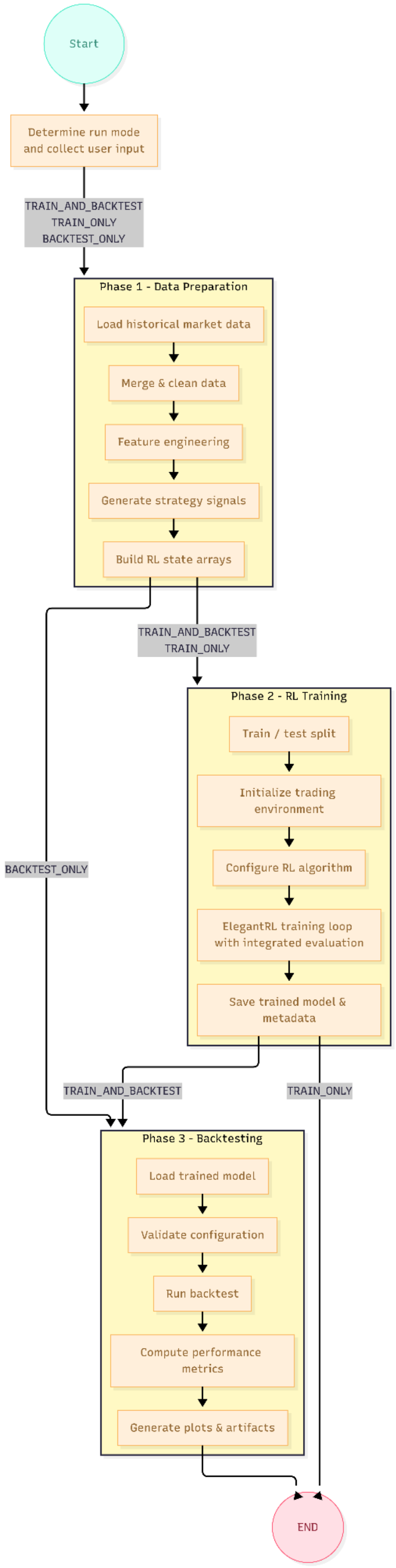

Figure 2: System Flow Diagram for RL Training and Evaluation

3.2.4. Technologies and Tools

The training and evaluation sub-project was developed in Python 3.11 using the PyCharm IDE. At its core, the system relies on ElegantRL, which serves as the primary reinforcement learning training framework.

ElegantRL is an open-source deep reinforcement learning (DRL) library implemented in PyTorch, designed for efficient and scalable training of modern reinforcement learning algorithms. The library emphasizes a lightweight architecture while supporting advanced training paradigms, including parallel environment interaction, policy evaluation and modular training orchestration.

ElegantRL provides robust implementations of policy-gradient and actor-critic algorithms, most notably PPO and SAC, which are the two algorithms supported by the proposed system. In this project, ElegantRL is responsible for executing the full reinforcement learning training loop, managing the interaction between the policy and the custom trading environment and performing periodic evaluation during training.

During the backtesting phase, trained policies produced by ElegantRL are executed in inference mode and evaluated on historical market data using identical environment dynamics and constraints, enabling consistent and reproducible performance assessment. [12]

Additional supporting libraries and tools used throughout the training, data processing and evaluation pipeline are detailed in **Appendix D**.

3.3. Sub-Project II: Client-Server Trading Execution System

Development status: A proof-of-concept (POC) was developed to design and prototype the system's continuous execution mechanism. The full server and client components are planned to be implemented as part of **Phase B.**

3.3.1. Overview

The client-server sub-project is responsible for executing the trained reinforcement learning agent in a live trading environment and serving as the operational runtime layer of the system.

The system is designed according to a client-server architecture with a clear separation of responsibilities. The server component encapsulates all trading logic and execution responsibilities, while the client component provides a graphical user interface for monitoring, management and interaction by end users and administrators. The server is intended to be deployed in a cloud environment (AWS) in order to support uninterrupted 24/7 operation and enable system scalability, which are fundamental requirements for Bitcoin trading. At runtime, the server continuously executes the selected trained reinforcement learning agent based on live market data and places trades through a direct connection to the Kraken[11] cryptocurrency exchange.

[11] **Kraken** is a widely used and reliable cryptocurrency exchange, chosen for its competitive fee structure and regulatory alignment suitable for users operating under Israeli regulations.

In addition to the trading execution logic, the server manages all backend system components, including databases for users, trained models, executed trades, market data and system logs. It also handles communication with the client application and integrates with external services required for secure credential storage and system monitoring.

The client application is implemented as a desktop graphical user interface that supports two user roles: end users and administrators. The user interface is designed to be clear, financial-oriented, and minimalistic, with an emphasis on usability and transparency. End users are required to independently create a trading account on the Kraken exchange and then register within the system to link their account. Once connected, users can allocate funds for algorithmic trading and activate or deactivate the trading process. User interaction is intentionally kept simple. Apart from selecting an investment amount and managing trading activation, users are not required to configure strategies or participate in trading decisions, and no prior trading knowledge is assumed.

Transparency is maintained by presenting users with detailed information about the trained agent, including historical training performance, metrics and visualizations. During live trading, users can observe executed trades on an interactive chart and track real-time performance metrics. A dedicated help and learning section provides explanations about the system and the trading strategies used.

Administrative users are provided with additional capabilities, including viewing all trained agents uploaded from the training sub-project, reviewing their performance, and selecting which agent is active for live trading. Administrators can upload trained agents to the server using a dedicated interface, view aggregated system statistics, and manage registered users without accessing sensitive information.

Across both server and client components, the system is designed with strong emphasis on security, reliability, and risk control. User authentication is protected using one-time password (OTP) mechanisms, sensitive credentials such as exchange API keys are stored using an external secure service, and system alerts are generated in response to critical events such as server failures. In addition, protective constraints are applied to trading decisions to mitigate the impact of irrational or extreme actions produced by the reinforcement learning agent.

3.3.2. Requirements

The Client-Server Trading Execution System is required to satisfy a comprehensive set of functional and non-functional requirements that define its operational behavior, system interfaces and quality attributes. These requirements cover the execution of live trading decisions, user and administrator interactions, data management, system communication and integration with external services such as cryptocurrency exchanges and secure credential storage.

In addition, non-functional requirements address critical system qualities including security, reliability, availability, performance and observability, which are essential for continuous 24/7 operation in a live trading environment.

A complete and structured specification of the functional and non-functional requirements for this sub-project is provided in **Appendix E**.

3.3.3. Continuous Execution Mechanism

The system is driven by a scheduler-based tick mechanism that serves as a periodic trigger for executing a complete decision-making cycle. Each tick initiates a fixed sequence of operations that processes new market information and produces trading actions.

At the beginning of each tick, the system retrieves only newly available market data, including OHLCV prices, technical indicators and market turbulence signals such as the VIX. This data is incorporated into a fixed-size rolling window, ensuring that the system maintains a bounded and up-to-date market view while preventing uncontrolled data growth.

The updated data is then prepared for strategy-level processing. Market data is aligned to the specific timeframes and lookback windows required by each strategy. For every strategy, the system checks whether the current tick matches the strategy's configured execution timeframe. If so, the strategy is executed and produces a new decision, otherwise, the most recent decision of the strategy is reused for the current tick.

All strategy decisions are converted into a one-hot encoded representation. In this representation, each possible decision is mapped to a binary vector, providing the reinforcement learning agent with a clear and discrete input format.

The system state vector is then constructed by combining the market data and the one-hot encoded strategy decisions. This state is normalized and passed to the trained reinforcement learning agent.

The trained reinforcement learning agent produces two continuous control values that describe the desired trading direction, level of exposure, and stop-loss behavior at a system level. These outputs do not represent concrete orders.

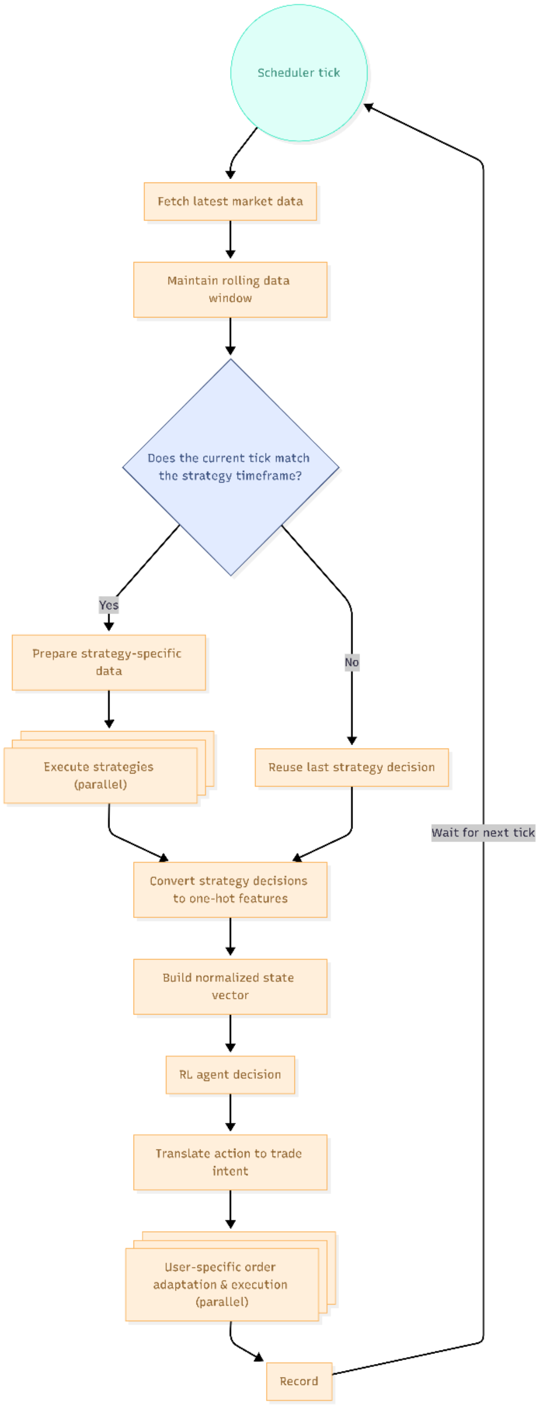
Scheduler tick
Fetch latest market data
Maintain rolling data window
Does the current tick match the strategy timeframe?
Yes
No
Prepare strategy-specific data
Execute strategies (parallel)
Reuse last strategy decision
Wait for next tick
Convert strategy decisions to one-hot features
Build normalized state vector
RL agent decision
Translate action to trade intent
User-specific order adaptation & execution (parallel)
Record

Figure 3: Continuous per-tick execution mechanism of the trading system

The agent output is first interpreted as a trade intent, which defines how the system should act in the market. This trade intent is then adapted on a per-user basis by applying user-specific parameters, such as allocated capital, to derive concrete order sizes. The resulting orders are executed on the exchange in parallel. At the end of the tick, all relevant outcomes are recorded.

3.3.4. Technologies and Tools

The client-server sub-project is implemented using a set of technologies that support continuous execution, secure communication, scalable deployment and interactive user interfaces. Shared libraries and frameworks used across the system are documented centrally in Appendix D.

Server-Side:

The server-side components are implemented in Python 3.11, which serves as the primary language for the execution logic and trading infrastructure.

- **Django-** serves as the backend framework for implementing server-side logic, user management, and APIs.
- **PostgreSQL 16 with TimescaleDB extension-** a relational SQL database used for persistent storage of users, trades, market data, and system logs. PostgreSQL provides a reliable and well-established data model for structured system data, while the TimescaleDB extension enables efficient handling and querying of time-series data generated by the trading system.
- **APScheduler-** used to implement the scheduler-driven tick mechanism that triggers periodic execution of the trading workflow.

Client-Side:

The client application is implemented in Java as a Maven-based project and developed using the IntelliJ IDEA IDE, with the specific Java version to be determined in Phase B.

- **JavaFX-** a Java-based framework for building rich desktop applications, used in this project to implement the graphical user interface for end users and administrators.

DevOps and Infrastructure:

- **Docker-** used for containerizing server-side components.
- **Kubernetes-** used for orchestrating and managing containerized services.
- **AWS-** serves as the cloud infrastructure platform for deployment and operation.
- **Argo CD-** used to manage continuous deployment workflows.
- **Terraform-** used to define and provision cloud resources programmatically using configuration files, enabling cloud infrastructure to be managed as code rather than through manual configuration via the AWS management interface.
- **Prometheus-** used for system monitoring and metric collection.

Communication:

- **HTTPS-** used for secure client-server communication.
- **WSS (Secure WebSockets)-** used for real-time bidirectional communication.

Additional supporting libraries and specific implementation details will be selected and finalized during Phase B, as the server and client components are further developed.

3.4. Sub-Project III: PineScript to Python Strategy Converter

Development status: The feasibility of an AI agent-based workflow for strategy conversion has been evaluated. The full implementation of this sub-project is planned as part of **Phase B.**

3.4.1. Overview

This sub-project addresses the challenge of integrating a large ecosystem of existing open-source trading strategies into the system. Many widely used strategies are implemented in TradingView's PineScript language, which is not directly compatible with the Python-based training and execution infrastructure used in this project. The goal is to convert PineScript strategies into Python implementations compatible with the system's internal strategy interface.

The conversion process is designed as a controlled, AI agent-based workflow that emphasizes correctness and safety. Rather than performing bulk or fully automated conversions, the system operates on a strategy-by-strategy basis, ensuring that each converted strategy is carefully validated before being introduced into the system. Human oversight is a core design principle, with explicit developer approval required at the final stage before a converted strategy is accepted and deployed.

The architecture is based on a multi-agent model coordinated by a central orchestrator agent. Specialized sub-agents are responsible for distinct aspects of the conversion process, including transpiling PineScript code into Python, validating syntactic and semantic correctness, generating automated tests, and integrating the resulting strategy into the existing codebase. The conversion preserves the original strategy logic while adapting it to a standardized Python structure.

Converted strategies are implemented as subclasses of a common "BaseStrategy" interface, which defines a unified structure and behavior expected by the training and execution systems. This ensures consistency across strategies and enables their reuse across both the reinforcement learning training engine and the live trading execution environment.

Quality assurance is enforced through automated validation, testing, and backtesting, with the workflow designed to fail safely and require developer intervention when issues are detected.

Upon successful completion, converted strategies are versioned and prepared for integration into the project's source repository through a controlled deployment process. This approach allows the system to gradually expand its strategy pool while maintaining reliability, traceability, and compatibility with the broader architecture.

3.4.2. Requirements

The PineScript to Python Strategy Converter sub-project is designed to satisfy a set of functional and non-functional requirements that ensure correct, safe and controlled conversion of trading strategies. These requirements define the supported inputs, AI agent-based conversion workflow, validation and testing steps, deployment constraints and developer approval points, as well as key quality attributes such as reliability, compatibility and testability.

A complete and structured specification of these requirements is provided in **Appendix F**.

3.4.3. Planned Flow for Conversion Workflow

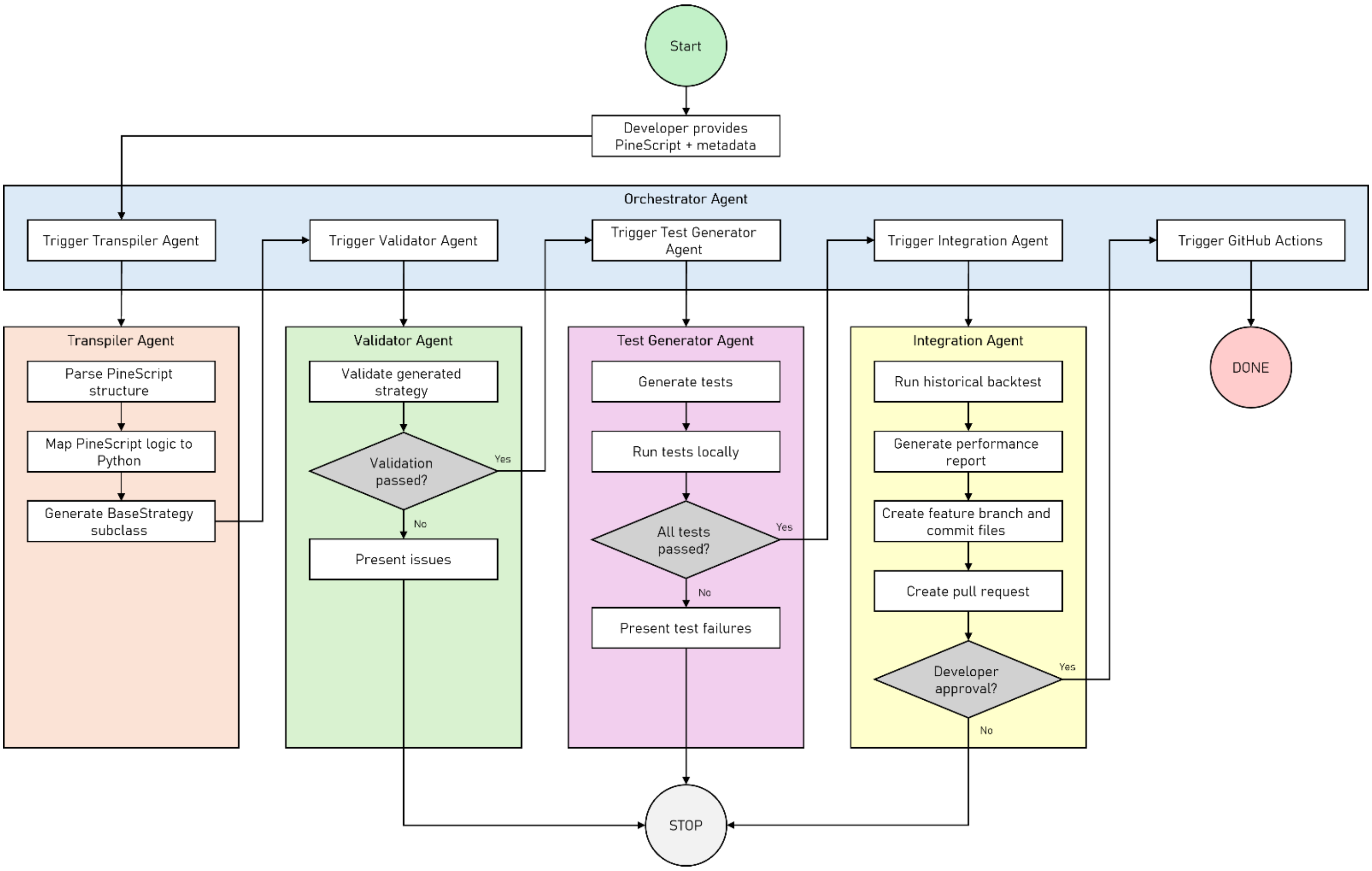

Figure 4: Agent-based strategy conversion workflow from PineScript to Python

The conversion process is coordinated by an orchestrator agent that manages the execution order of specialized sub-agents. The workflow begins once a developer submits PineScript code along with basic strategy metadata such as the strategy name, timeframe, and lookback window.

The transpiler agent parses the PineScript structure, maps PineScript constructs and indicators to their Python equivalents and generates a Python strategy that implements the project's BaseStrategy.

The generated strategy is then passed to the validator agent, which performs syntactic, semantic, and interface compliance checks. This includes verification of correct imports, preservation of trading logic, and detection of issues such as lookahead bias. If validation fails, the process is halted and the detected issues are reported.

Upon successful validation, the test generator agent produces and executes automated tests, including unit tests for indicator calculations, integration tests for full strategy execution, and edge case tests. The workflow proceeds only if all tests pass.

The integration agent then performs a historical backtest of the strategy and generates a performance report. If the results are acceptable, the agent creates a dedicated GitHub feature branch, commits the strategy and test files, updates the strategy registry, and opens a pull request.

The workflow concludes with explicit developer approval of the pull request. Once approved, the strategy is ready for further integration through the project's continuous integration pipeline.

3.4.4. Technologies and Tools

The strategy conversion sub-project is implemented as a Python 3.11 project developed in the PyCharm integrated development environment and relies on agent-based code generation and validation.

Claude Code:

- Serves as the primary agent execution environment for the strategy conversion workflow.
- Selected due to its strong capabilities in structured code generation, multi-step reasoning, and strict adherence to explicit instructions, which are essential for preserving trading logic during conversion.
- Agent behavior is defined through a dedicated ".claude" directory within the project repositor.
- The ".claude/agents" directory contains Markdown files that specify the role, responsibilities, and constraints of each agent.
- The ".claude/skills" directory defines shared rules and conventions that apply across all agents, enabling consistent outputs and controlled behavior.
- Developer interaction is performed through the Claude Code command-line interface, where the main agent is invoked directly from the terminal.

GitHub MCP:

- GitHub Model Context Protocol (MCP)[12] is used to provide Claude Code with direct, programmatic access to GitHub repositories from the Claude command-line environment.
- This integration allows agents to interact with the repository without manual developer intervention.
- Supports repository-level operations such as creating feature branches, committing generated strategy and test files, and opening pull requests.
- Ensures that converted strategies are version-controlled and integrated into the development workflow in a traceable and auditable manner.

Converted strategies rely on the same Python-based ecosystem used across the project, including technical analysis and strategy-related libraries documented in Appendix D.

[12] **Model Context Protocol (MCP):** a protocol that enables large language models (LLMs) to interact with external tools and systems through a structured and controlled interface, allowing models to perform actions beyond text generation while maintaining traceability and execution constraints.

3.5. Integrated System Architecture

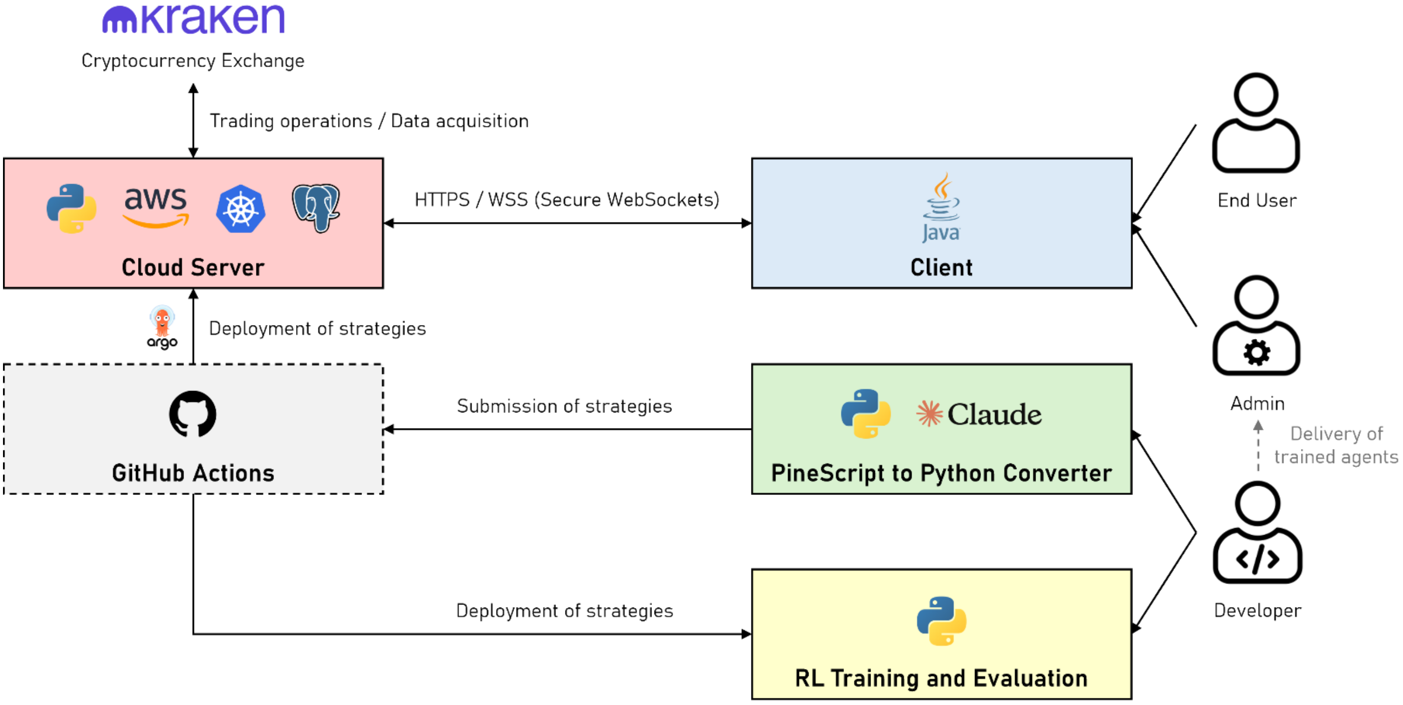

Figure 5: High-level integrated architecture of the algorithmic trading system and its main components

The integrated system architecture combines the three sub-projects into a single operational trading platform with clear responsibility boundaries. The cloud server acts as the central runtime component of the system. It maintains persistent databases, handles all user-related operations, and communicates with the cryptocurrency exchange for market data acquisition and trade execution. All trading activity and system state are managed on the server side, enabling continuous 24/7 operation.

Strategy development is performed externally by the developer. Trading strategies are converted using the strategy conversion sub-project. Once validated, these strategies are automatically integrated into both the server codebase and the RL training environment through version-controlled pipelines based on GitHub Actions. This ensures that the same strategy implementations are consistently available for offline training and live execution.

Reinforcement learning agents are trained and evaluated within the training sub-project. Upon completion, the developer produces a set of artifacts that include the trained agent along with its performance metrics and evaluation outputs. These artifacts are delivered to the system administrator, who reviews the results and uploads the selected agent to the cloud server through the administrative GUI. Only agents explicitly approved and deployed by the administrator are activated for live trading.

The client application communicates securely with the cloud server and serves as the primary interface for both end users and administrators. End users interact with the system at a high level, while administrators are provided with additional capabilities for managing deployed agents and monitoring system operation. Together, these components form a modular and controlled architecture that supports strategy evolution, offline learning, and reliable live trading within a unified system.

4. Development and Testing Methodology

4.1. Development Workflow and Project Management

The development process of the project is centered around a structured, repository-driven workflow using **GitHub** as the primary collaboration and integration platform.

All project assets are hosted under a public GitHub organization named **"AlgoTrading-Capstone"**, which encapsulates the different system components as separate repositories.

In addition to the code repositories, a dedicated repository named **"platform"** is used as a central coordination layer. This repository contains high-level project artifacts, including development tasks, planning documents and CI/CD-related workflows. By separating platform-level management concerns from implementation repositories, the project maintains a clear distinction between system governance and executable components.

GitHub Projects:

Task management and collaborative planning are handled using GitHub Projects, which serve as the primary mechanism for maintaining the backlog, planning iterations, and tracking the overall development roadmap. Tasks are organized into structured boards and timelines, enabling clear visibility into progress, priorities, and upcoming milestones. Where applicable, automation rules and workflows are employed to synchronize issues, pull requests, and project states, thereby reducing manual coordination overhead and improving process consistency.

GitHub Actions:

Continuous Integration (CI) is implemented using GitHub Actions, which are configured to automatically validate changes pushed to the repositories. These workflows execute automated checks such as linting, test execution, and build validation, ensuring that code quality, correctness, and system stability are preserved throughout the development lifecycle.

GitHub Issues:

Issue tracking is performed using GitHub Issues, providing a unified mechanism for documenting defects, enhancements, and technical discussions across the project. Issues are used to capture both functional and technical concerns and are integrated into the broader project management workflow. The structure, categorization, and lifecycle of issues are described in detail in a dedicated section later in this document.

In addition to conventional development and project management practices, the project made extensive and structured use of artificial intelligence-based tools throughout the research, design, development and testing phases. These tools were integrated as engineering assistants to support productivity, knowledge exploration and implementation, while maintaining human oversight over all design and validation decisions. A detailed description of the AI tools used and their role within the development workflow is provided in **Appendix G**.

4.2. Testing Process and Validation Methodology

The testing process was designed to verify that the system behaves as intended and meets its functional requirements. Due to the system's modular structure, different testing methods are applied to different parts of the system.

4.2.1. General Testing Philosophy

Testing is integrated as a continuous activity throughout the development lifecycle and follows Agile principles. Rather than treating testing as a final phase, quality assurance is embedded into daily development through automated verification and iterative validation. Each code change is expected to satisfy a clearly defined "Definition of Done" (DoD), which includes successful execution of automated test suites, compliance with code quality standards, completed code reviews, and up-to-date documentation.

Given the academic constraints of the project, the testing scope prioritizes automated unit and integration testing for core logic and backend components, complemented by manual acceptance testing for user-facing graphical interfaces. This approach ensures high development velocity while maintaining confidence in system correctness.

4.2.2. Testing Infrastructure and Continuous Integration

The testing infrastructure is tightly coupled with the project's DevOps workflow. Continuous Integration is enforced through automated pipelines, ensuring that all changes are validated before merging. A strict "build breaker" policy is enforced: code cannot be merged into the main branch unless all CI checks pass successfully. This guarantees that the shared codebase always remains in a deployable and stable state.

4.2.3. Testing Sub-Project I: RL Training and Evaluation

Testing of the reinforcement learning (RL) component explicitly distinguishes between verification and validation, reflecting the difference between software correctness and model performance.

Verification focuses on deterministic components of the training pipeline and is implemented using automated unit tests. These tests validate technical indicator calculations against known reference values, ensure correct behavior of the reward function for specific state transitions, and verify the mechanics of the trading environment, such as portfolio updates, leverage constraints, and stop-loss handling.

Validation addresses the quality and effectiveness of the trained model rather than code correctness. The primary validation method is backtesting on historical market data. The model is evaluated across multiple time periods, including high-volatility regimes, while strictly separating training and evaluation datasets to prevent look-ahead bias. Performance metrics such as cumulative return, Sharpe ratio, and maximum drawdown are compared against baseline strategies, including a buy-and-hold benchmark.

4.2.4. Testing Sub-Project II: Client-Server Trading Execution System

For the client-server execution system, testing is divided into backend logic, frontend behavior, and security aspects.

Backend testing is implemented using automated unit and integration tests. External dependencies, such as cryptocurrency exchange APIs, are mocked to enable deterministic testing without real financial transactions.

Frontend testing is performed through manual user acceptance testing. Predefined user scenarios derived from the functional requirements are executed. Test results and observed issues are documented using GitHub Issues to maintain traceability.

Security-related tests verify that sensitive data, such as API keys, are stored encrypted at rest, that authentication mechanisms cannot be bypassed, and that user inputs are properly validated and sanitized.

4.2.5. Testing Sub-Project III: PineScript to Python Strategy Converter

Testing of the strategy conversion component focuses on logical correctness rather than exact numerical equivalence. Verification ensures that converted strategies conform to the required interface and execute correctly, while validation relies on controlled testing and visual inspection to confirm consistency with the original strategy logic.

4.2.6. Requirements Traceability and Verification Evidence

To ensure traceability between requirements and testing activities, GitHub Projects is used as the central coordination tool. Functional and non-functional requirements are explicitly mapped to development tasks and issues using a labeling scheme. Each pull request must reference the relevant requirement-linked issue and include appropriate tests.

4.3. Expected Challenges

4.3.1. Budget Constraints

Challenge: developing and operating the system involves potential costs related to cloud infrastructure (such as AWS compute and storage), paid AI tools (including ChatGPT, Gemini, and Claude subscriptions), development environments (IDEs) and third-party software. Without careful planning, these costs may become significant.

Mitigation: to address this challenge, the project prioritizes the use of free resources available to students. This includes leveraging the **GitHub Student Developer Pack**, free student access to Gemini and AWS student credits of up to 200$. Whenever possible, open-source tools and publicly available information are preferred. In addition, computationally intensive tasks such as model training are performed on personal machines rather than cloud infrastructure to further reduce operational costs.

4.3.2. Time Constraints

Challenge: the project is developed alongside academic coursework, employment obligations, and personal commitments. Balancing these responsibilities while maintaining steady progress poses a significant time management challenge.

Mitigation: this challenge is addressed through structured time management and task planning. Project management and scheduling tools are used to track progress and prioritize tasks. AI tools are also leveraged to improve productivity, reduce manual effort, and accelerate research, development, and documentation activities.

4.3.3. Development Complexity

Challenge: this project represents the first time the team is working on a system of this scale, combining multiple technologies, architectural layers and development paradigms. The use of unfamiliar technologies further increases the complexity and learning curve.

Mitigation: to manage this complexity, the system is decomposed into smaller, well-defined components. Tasks are distributed according to each team member's strengths and areas of responsibility. In parallel, additional learning resources are used to close knowledge gaps, including self-paced courses such as a dedicated DevOps course. AI tools are also used as supportive aids.

4.3.4. Limited Domain Knowledge in Trading and Cryptocurrencies

Challenge: at the outset of the project, the team had limited prior experience and practical knowledge in algorithmic trading and cryptocurrency markets, which are central to the system's domain.

Mitigation: to address this gap, the team completed a dedicated Udemy course focused on algorithmic Bitcoin trading. This provided essential background knowledge, practical examples, and domain-specific terminology required to design, implement, and evaluate the trading system more effectively.

5. Project Success Metrics

The success of the project is evaluated using a set of quantitative **Key Performance Indicators (KPIs)** that measure trading performance, risk exposure, system behavior and operational reliability across the different system components and execution environments. These metrics provide an objective and consistent basis for validating system correctness, comparing trained agents and assessing readiness for live operation.

A complete specification of all KPIs, including metric definitions, target values and evaluation methods, is provided in **Appendix H**.

6. References

[1] Mordor Intelligence. (2024). **Algorithmic trading market & share analysis- growth, trends, and forecast (2025 - 2030).** https://www.mordorintelligence.com/industry-reports/algorithmic-trading-market

[2] Sebastião, H., & Godinho, P. (2021). **Forecasting and trading cryptocurrencies with machine learning under changing market conditions.** Financial Innovation, 7(1), Article 3. https://doi.org/10.1186/s40854-020-00217-x

[3] Salman, O., Melissourgos, T., & Kampouridis, M. (2025). **A genetic algorithm for the optimization of multi-threshold trading strategies in the directional changes paradigm.** Artificial Intelligence Review, 59, Article 2. https://doi.org/10.1007/s10462-025-11419-z

[4] Baron, M., Brogaard, J., Hagströmer, B., & Kirilenko, A. (2019). **Risk and return in high-frequency trading.** Journal of Financial and Quantitative Analysis, 54(3), 993-1024. https://doi.org/10.1017/S0022109018001096

[5] Treleaven, P., Galas, M., & Lalchand, V. (2013). **Algorithmic trading review.** Communications of the ACM, 56(11), 76-85. https://doi.org/10.1145/2500117

[6] Zhang, Z., Zohren, S., & Roberts, S. (2020). **Deep reinforcement learning for trading.** The Journal of Financial Data Science, 2(2), 25-40.https://doi.org/10.3905/jfds.2020.1.030

[7] Sutton, R. S., & Barto, A. G. (2018). **Reinforcement learning: An introduction**. (2nd ed.). MIT Press.

[8] Liu, F., Li, Y., Li, B., Li, J., & Xie, H. (2021). **Bitcoin transaction strategy construction based on deep reinforcement learning.** arXiv preprint arXiv:2109.14789. https://doi.org/10.48550/arXiv.2109.14789

[9] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). **Soft actor-critic: Off-policy deep reinforcement learning with a stochastic actor.** International Conference on Machine Learning (ICML), 1861–1870. PMLR. https://doi.org/10.48550/arXiv.1801.01290

[10] Liu, X. Y., Yang, H., Chen, Q., Zhang, R., Yang, L., Xiao, B., & Wang, C. D. (2020). **FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance.** Deep RL Workshop, NeurIPS 2020. https://doi.org/10.48550/arXiv.2011.09607

[11] Sun, S., Wang, R., & An, B. (2023). **Reinforcement learning for quantitative trading.** ACM Transactions on Intelligent Systems and Technology, 14(3), Article 44. https://doi.org/10.1145/3582560

[12] ElegantRL Contributors. (n.d.). **ElegantRL documentation.** https://elegantrl.readthedocs.io/en/latest/index.html#

[13] Liu, X.-Y., Xiong, Z., Zhong, S., Yang, H., & Walid, A. (2018). **Practical deep reinforcement learning approach for stock trading.** arXiv preprint arXiv:1811.07522. https://doi.org/10.48550/arXiv.1811.07522

7. Appendices

Appendix A- Example Financial Trading Strategy- Moving Average Crossover

This appendix presents a simple example of a financial trading strategy in order to illustrate the basic concept of an algorithmic trading strategy. While the strategy is generic and can be applied to various financial instruments, it is also commonly used in cryptocurrency markets, including Bitcoin.

The Moving Average Crossover strategy is a trend-following approach based on two moving averages calculated over different time horizons. A moving average represents a smoothed price series, computed as the average of past prices over a defined window, and is used to reduce short-term noise while highlighting the underlying market trend.

In this strategy, a short-term moving average (e.g., 20 periods, where a period corresponds to one price bar in the selected timeframe) reacts more quickly to recent price changes, while a long-term moving average (e.g., 50 periods) reflects the broader market direction. A bullish signal (often interpreted as a buy indication) is generated when the short-term moving average crosses above the long-term moving average. Conversely, a bearish signal (sell indication) occurs when the short-term moving average crosses below the long-term moving average.

The rationale behind this strategy is that such crossovers indicate a shift in market momentum, suggesting a potential change in trend direction. Although simple, this example demonstrates the core idea of a trading strategy: transforming historical market data into explicit, rule-based decisions that can be executed systematically.

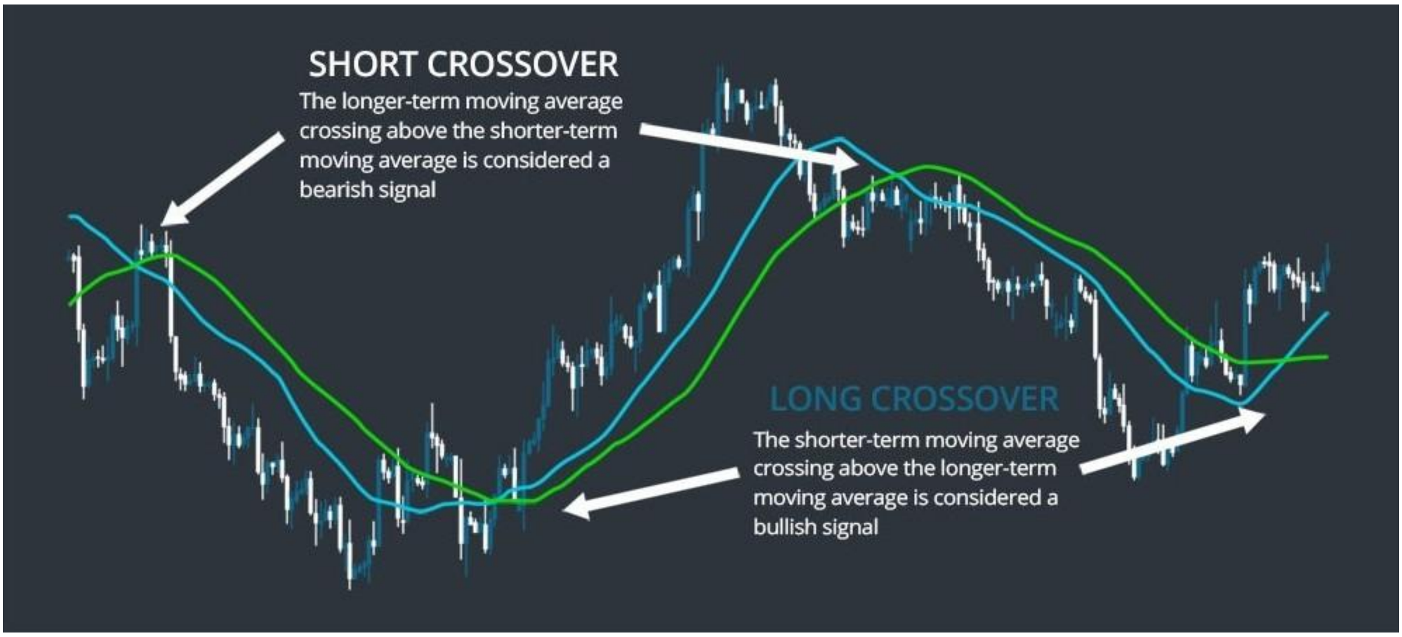

Figure A.1: Illustration of a moving average crossover trading strategy using short-term and long-term moving averages.

Appendix B- Training Configuration Parameters

This appendix summarizes the key configuration parameters that control the reinforcement learning training process, trading environment constraints and feature construction.

These parameters define the experimental search space and enable systematic, reproducible evaluation of trained agents under varying modeling assumptions.

B.1. Reinforcement Learning Configuration

The system supports multiple reinforcement learning algorithms, each with configurable hyperparameters.

Algorithm Selection:

- **PPO (Proximal Policy Optimization)-** on-policy, stable updates, suitable as a baseline.
- **SAC (Soft Actor-Critic)-** off-policy, higher sample efficiency, increased variance.

Core Hyperparameters:

- **Discount factor-** controls the trade-off between short-term and long-term rewards.
- **Learning rate-** determines the magnitude of updates to the policy and value networks.
- **Neural network architecture-** number and size of hidden layers for actor and critic networks.
- **Total training steps-** maximum number of environment interactions across all episodes.

Algorithm-Specific Parameters:

- **PPO:** rollout horizon, number of optimization epochs per rollout, clipping ratio, GAE parameters, entropy regularization.
- **SAC:** replay buffer size, batch size, target network update rate, entropy temperature (fixed or automatically tuned).

B.2. Trading Environment and Risk Constraints

The trading environment enforces realistic portfolio and execution constraints to prevent degenerate or unsafe policies.

- **Initial capital-** starting portfolio balance at the beginning of each episode.
- **Leverage limit-** maximum allowable exposure relative to current equity.
- **Maximum position size-** hard cap on absolute BTC exposure.
- **Exposure dead-zone-** minimum change in target exposure required to trigger a trade.
- **Transaction costs-** proportional trading fees applied to executed orders.
- **Slippage model-** stochastic price impact applied to market orders.
- **Stop-loss bounds-** minimum and maximum allowable stop-loss distance relative to entry price.

B.3. Reward Function Design

The reward function defines the optimization objective for the agent.

- **Reward formulation-** selectable reward definitions (e.g., log-returns, asymmetric rewards).
- **Downside penalty weight-** controls the relative punishment of negative returns compared to positive gains.
- **Risk-free rate-** reference rate used in risk-adjusted performance metrics.

B.4. Market Data and Feature Selection

The observation space is constructed from market data and derived features.

- **Market timeframe-** candle resolution used for environment stepping (e.g., 15-minute bars).
- **Technical indicators-** momentum, trend, and volatility indicators included in the state vector.
- **Risk and stress indicators-** optional features such as market turbulence measures.
- **External market signals-** exogenous time series (e.g., volatility indices) aligned to the trading timeframe.
- **Train-test split ratio-** proportion of historical data allocated to training versus evaluation.

B.5. Strategy Integration Parameters

In addition to raw market features, the system can incorporate signals from rule-based trading strategies.

- **Strategy enablement flag-** controls whether strategy signals are included in the state representation.
- **Strategy set selection-** configurable list of enabled strategies.
- **Signal encoding-** each strategy contributes a fixed-size one-hot encoded signal vector.
- **Parallel strategy processing-** maximum number of workers used for strategy signal generation.

Appendix C- Detailed Requirements for Sub-Project I: RL Training and Evaluation

C.1. Functional Requirements (FR)

Req. ID	Req. Description
Interface and Execution Modes	
FR-1	The system shall provide a Command-Line Interface (CLI) to operate the application.
FR-2	The system shall execute a "training combined with backtesting" mode to train an agent and immediately evaluate it.
FR-3	The system shall execute a "training only" mode to train reinforcement learning agents without immediate backtesting.
FR-4	The system shall execute a "backtesting only" mode to evaluate previously trained models on historical data.
FR-5	The system shall allow the user to specify the date range for training data and backtesting data.
FR-6	The system shall allow configuration of the cryptocurrency exchange source for market data retrieval.
FR-7	The system shall allow configuration of the market data timeframe.
Data Processing	
FR-8	The system shall ingest historical Bitcoin OHLCV data from cached local storage or by downloading from exchanges.
FR-9	The system shall enrich historical data with configurable technical indicators and risk-related features.

FR-10	The system shall enrich historical data with signals from multiple trading strategies, represented as part of the state.
Simulation Environment	
FR-11	The system shall track the agent's account balance and total portfolio equity over time.
FR-12	The system shall track open positions, including their direction and size.
FR-13	The system shall simulate transaction costs by applying trading fees to executed orders.
FR-14	The system shall simulate slippage behavior during trade execution.
FR-15	The system shall enforce leverage constraints to limit the agent's maximum market exposure.
FR-16	The system shall enforce position limits to restrict the maximum holdings.
FR-17	The system shall execute dynamic stop-loss mechanisms to close positions when specific levels are reached.
Configuration and Hyperparameters	
FR-18	The system shall load training parameters from a centralized configuration file.
Outputs and Logging	
FR-19	The system shall save the trained reinforcement learning model to a file upon completion.
FR-20	The system shall generate metadata files describing the specific configuration and environment used for the session.
FR-21	The system shall generate step-level logs capturing the state, action, and reward at each time step.
FR-22	The system shall generate trade-level logs detailing individual trade executions and position changes.
FR-23	The system shall calculate quantitative performance metrics summarizing returns and risk characteristics.
FR-24	The system shall generate visual performance comparison charts benchmarking the agent against baseline strategies.
FR-25	The system shall validate compatibility between a trained model and backtesting data.

C.2. Non-Functional Requirements (NFR)

Req. ID	Req. Description
Availability	
NFR-1	The system shall download only missing data segments from exchanges, avoiding redundant retrieval of existing data.
NFR-2	The system shall implement retry logic with exponential backoff for external data retrieval failures.
Usability	
NFR-3	The CLI shall display progress indicators showing elapsed time, estimated remaining time and current processing phase.
NFR-4	The CLI shall use color-coded output to distinguish between informational messages, warnings and errors.
Testability	
NFR-5	The system architecture shall enable unit testing of individual components without dependencies on external services.
NFR-6	The system shall support reproducible training runs through configurable random seeds.
Maintainability	
NFR-7	The system shall be architected to strictly decouple configuration data from algorithmic logic.
Portability	
NFR-8	The processed training data cache shall be transferable between development machines without requiring reprocessing.
NFR-9	The system shall be installable on different machines with minimal environment configuration.
NFR-10	The system shall support ingestion of externally provided raw OHLCV data files as an alternative to exchange download.
Traceability	
NFR-11	All training artifacts shall be organized within a structured results directory.
NFR-12	The system shall create a directory for each session, named using a composite of the agent name and machine identifier.
NFR-13	The system shall create a sub-directory within the session folder for each backtest, named using a timestamp.
Performance	
NFR-14	The system shall execute trading strategy signal generation in parallel.
NFR-15	The system shall support GPU acceleration (CUDA) for neural network training when compatible hardware is available.

Appendix D- Shared Libraries and Tools

This appendix summarizes software libraries and tools that are shared across multiple sub-projects within the system.

These components support numerical computation, time-series processing, feature engineering, technical analysis, market data access, data storage, logging, visualization and developer-facing interfaces.

- **NumPy-** used for numerical computations and array-based data structures.
- **Pandas-** used for time-series manipulation, data cleaning, feature engineering, and alignment of market data and strategy signals.
- **PyTorch-** serves as the underlying deep learning framework for neural network models and reinforcement learning components.
- **CCXT-** provides a unified interface for accessing cryptocurrency exchange APIs and retrieving historical market data.
- **Yfinance-** used to obtain external financial market data that can be incorporated into the feature set when required.
- **TA-Lib-** used for computing technical indicators during feature engineering.
- **Pyarrow-** enables efficient storage and loading of large datasets using the Parquet columnar file format.
- **Fastparquet-** provides an alternative Parquet backend optimized for performance and compatibility.
- **Loguru-** used for structured and configurable logging throughout all pipeline stages to support monitoring and debugging.
- **Rich-** used to enhance the command-line interface with formatted output and improved usability.
- **Matplotlib-** used to generate performance visualizations and evaluation plots during backtesting.

Appendix E- Detailed Requirements for Sub-Project II: Client-Server Trading Execution System

E.1. Functional Requirements (FR)

Req. ID	Req. Description
	User Authentication & Registration
FR-26	The system shall provide a user registration process.
FR-27	The system shall require email verification during the registration process.
FR-28	The system shall require users to configure OTP authentication during the registration process.
FR-29	The system shall authenticate users via OTP for all login attempts.
FR-30	The system shall prompt first-time users to connect their Kraken exchange account after initial login.
FR-31	The system shall provide a redirect link to Kraken's website for users who do not have an existing exchange account.
FR-32	The system shall securely collect and validate Kraken API credentials from users.
	User Trading Operations
FR-33	The system shall allow users to specify the amount of funds to allocate for algorithmic trading.

FR-34	The system shall allow users to start algorithmic trading.
FR-35	The system shall allow users to stop algorithmic trading.
FR-36	The system shall allow users to add additional funds at any time.
FR-37	The system shall allow users to withdraw funds at any time.
FR-38	The system shall allow users to configure a recurring automatic profit withdrawal interval.
FR-39	The system shall allow users to export their complete trade history as a downloadable file.
FR-40	The system shall require user confirmation before executing irreversible operations.
User Monitoring & Information	
FR-41	The system shall display the agent's training metrics and performance charts.
FR-42	The system shall display real-time agent actions on an interactive price chart.
FR-43	The system shall display current live trading performance metrics.
FR-44	The system shall provide a help section explaining the system.
FR-45	The system shall display explanations for each trading strategy used by the active agent.
Admin Features	
FR-46	The system shall allow administrators to upload trained agent files.
FR-47	The system shall allow administrators to view all trained agents uploaded to the server.
FR-48	The system shall allow administrators to view performance metrics for each uploaded agent.
FR-49	The system shall allow administrators to select which agent is active for live trading.
FR-50	The system shall allow administrators to view registered users.
FR-51	The system shall allow administrators to view live trading statistics.
FR-52	The system shall support multiple administrator accounts with identical permissions.
Trading Engine	
FR-53	The system shall fetch real-time market data from Kraken exchange.
FR-54	The system shall compute technical indicators based on the fetched market data.
FR-55	The system shall fetch real-time VIX (Volatility Index) data from Yahoo Finance.
FR-56	The system shall enforce a configurable leverage limit that restricts maximum investment relative to user equity.
FR-57	The system shall enforce a configurable maximum position size as a percentage of user allocated funds.
FR-58	The system shall display the transaction fees charged by the crypto platform for each executed trade.
FR-59	The system shall allow hot-swapping of the active trading agent without system restart.
FR-60	The system shall execute a scheduled tick cycle at a configurable time interval.
FR-61	The system shall maintain a rolling window of historical OHLCV data by removing outdated records.
FR-62	The system shall determine strategy execution based on the current tick and configured timeframes.
FR-63	The system shall prepare and resample OHLCV data according to each strategy's required timeframe before execution.
FR-64	The system shall execute scheduled trading strategies.
FR-65	The system shall convert strategy decisions to one-hot encoded features.
FR-66	The system shall build a normalized state vector for the RL agent.
FR-67	The system shall pass the normalized state vector to the RL agent for decision making.
FR-68	The system shall translate the RL agent's actions into trade intent.
FR-69	The system shall adapt trade intent to user-specific orders based on each user's allocated funds.
FR-70	The system shall execute trade orders individually per user on the exchange.
FR-71	The system shall record all executed trades and decisions.
Communication	
FR-72	The system shall provide real-time data streaming to clients.
FR-73	The system shall send emergency alerts to administrators.
FR-74	The system shall export operational metrics to cloud-based monitoring services.
Database	
FR-75	The system shall store user account information.
FR-76	The system shall store all executed trade records.
FR-77	The system shall store configuration and metadata for all uploaded agents.
FR-78	The system shall store system logs and audit trails.

E.2. Non-Functional Requirements (NFR)

Req. ID	Req. Description
Security	
NFR-16	User API credentials shall be stored in a dedicated external secrets management service.
NFR-17	All client-server communications shall be encrypted using HTTPS/TLS.
NFR-18	The system shall log all authentication attempts and administrative actions.
NFR-19	The administrator user view shall exclude sensitive information such as API credentials and financial details.
Usability	

NFR-20	The user interface shall follow a minimalist financial dashboard design.
NFR-21	The user interface shall be operable without prior trading or cryptocurrency knowledge.
NFR-22	All interactive charts shall support zoom, pan and time range selection.
NFR-23	The agent file upload interface shall support drag-and-drop functionality.
NFR-24	The minimum interval for automatic profit withdrawal shall be one month.
Availability	
NFR-25	The trading engine shall operate continuously without scheduled downtime.
NFR-26	The system shall implement automatic recovery after unexpected failures.
NFR-27	Agent model files shall be stored in cloud storage.
Reliability	
NFR-28	The following operations shall be irreversible: fund allocation, trading start, trading stop, and fund withdrawal.
Observability	
NFR-29	Emergency alerts shall be delivered via email notification.
Data Requirements	
NFR-30	User registration shall collect the following fields: first name, last name, email, phone number, date of birth and country.
Performance	
NFR-31	Trading strategy execution shall be performed in parallel to minimize processing time.
NFR-32	User-specific order execution shall be performed in parallel to minimize processing time.

Appendix F- Detailed Requirements for Sub-Project III: PineScript to Python Strategy Converter

F.1. Functional Requirements (FR)

Req. ID	Req. Description
Input & Initialization	
FR-79	The system shall accept PineScript strategy code.
FR-80	The system shall accept metadata describing the PineScript strategy to be converted.
FR-81	The system shall include a Knowledge Base containing reference documentation for the conversion process.
AI Agents	
FR-82	The Orchestrator Agent shall control the conversion workflow and route tasks to sub-agents.
FR-83	The Transpiler Agent shall convert PineScript code to Python code.
FR-84	The Validator Agent shall validate the converted code.
FR-85	The Test Generator Agent shall create tests for the converted strategy.
FR-86	The Integration Agent shall handle verification and deployment.
Validation	
FR-87	The Validator Agent shall return control to the Orchestrator Agent when validation fails.
Test Generation	
FR-88	The system shall execute all generated tests and verify they pass before proceeding.
Strategy Verification	
FR-89	The system shall execute the converted strategy on historical data.
FR-90	The system shall verify the strategy produces valid trading signals.
FR-91	The system shall present verification results to the developer for approval before deployment.
Deployment	
FR-92	The Integration Agent shall deploy approved strategies to the target repository.
FR-93	The Integration Agent shall update the strategy registry with new strategy entries.
Developer Interaction	
FR-94	The system shall present detailed failure reports when validation or tests fail.
FR-95	The system shall prompt the developer for instructions when test execution fails.

F.2. Non-Functional Requirements (NFR)

Req. ID	Req. Description
Operational Constraints	
NFR-33	The system shall support strategies with timeframes of 15 minutes or longer.
Reliability	
NFR-34	Agents shall not implement automatic retry loops on failures.

NFR-35	All agent errors shall be logged with sufficient detail for developer diagnosis.
NFR-36	The converted code shall preserve the original PineScript logic exactly.
Integration	
NFR-37	Converted strategies shall be compatible with the training Engine.
NFR-38	Deployment shall create a feature branch and require Pull Request approval before merging.
NFR-39	Pull Request merge shall trigger the CI workflow automatically.
Usability	
NFR-40	The system shall support PineScript input via manual text entry.
NFR-41	The conversion process shall complete without manual intervention until the approval checkpoint.
NFR-42	The system shall support PineScript input via web scraping (optional future enhancement).
Data Requirements	
NFR-43	Strategy metadata shall include: strategy name, timeframe, and lookback period.
NFR-44	The system's Knowledge Base shall be initialized with reference documentation, including syntax mapping rules.
Compatibility	
NFR-45	Generated strategy classes shall inherit from BaseStrategy.
Testability	
NFR-46	Generated tests shall include: unit tests, integration tests and edge case tests.
NFR-47	Edge case tests shall cover: NaN values, empty data and insufficient candles.
NFR-48	Generated tests shall achieve a minimum of 95% code coverage.

Appendix G- Leveraging Artificial Intelligence (AI) Tools

In recent years, artificial intelligence-based tools have become an integral part of modern software engineering workflows. The rapid advancement of large language models and AI-assisted development platforms has significantly lowered the barrier for designing, implementing, and validating complex software systems. Projects at the scale and complexity of the proposed system-combining reinforcement learning, financial data pipelines, cloud-based services, and continuous deployment would have been extremely challenging to develop within a short timeframe by a small team using traditional methodologies alone.

G.1. Research and Knowledge Exploration

AI tools were extensively used during the research phase to support literature review and background exploration. Academic search platforms such as **Scholar Labs** and **Perplexity** were employed to identify relevant peer-reviewed papers. These tools enabled faster discovery.

To efficiently process and understand academic material, **NotebookLM** was used for structured review and summarization of selected papers. This allowed the extraction of key concepts, assumptions, and methodological insights while maintaining traceability to the original sources.

In addition, **Gemini** and **ChatGPT** Deep Research modes were leveraged for broader web-based exploration, including technical blogs, open-source documentation and industry practices.

G.2. System Design and Development

During the planning and implementation phases, AI tools were used as active engineering assistants rather than as automated code generators. **ChatGPT** and **Gemini** supported architectural reasoning and validation of design decisions. For hands-on development, **GitHub Copilot** and **Claude Code** were utilized to accelerate coding tasks.

To maintain consistent context across different AI engines and between both team members, shared Markdown (.md) files were continuously updated throughout the project. These documents captured architectural decisions, data schemas and constraints, ensuring that AI-assisted interactions remained aligned with the evolving system state and reducing context drift across tools.

G.3. Testing and Validation

Claude Code was used to assist in the automatic generation of unit tests based on existing function signatures and behavioral descriptions.

G.4. Operational Integration of AI Tools

AI tools were integrated into the operational workflow of the system. **Claude**-based agents were utilized to autonomously convert trading strategies, as described in the relevant sub-project.

Overall, the deliberate and structured use of AI tools enhanced productivity across all project stages, while human oversight remained central to design decisions, validation, and final responsibility for system behavior.

Appendix H- Key Performance Indicators (KPIs)

KPI ID	Metric	Target Value	Evaluation Method
A. Best Trained Agent Backtest Performance			
A-1	Excess Return vs. Buy & Hold	≥ 5 pp (percentage points)	The trained agent with the strongest overall backtest performance is selected for evaluation. KPI values are computed automatically as part of the backtesting process. The metrics are stored in metrics.json. **Evaluation Period:** Minimum of one year (backtest period).
A-2	Sharpe Ratio[13]	≥ 0.5	
A-3	Sortino Ratio[14]	≥ 0.7	
A-4	CAGR[15]	≥ 10% annualized	
A-5	Maximum Drawdown	≤ 30%	
A-6	Profit Factor[16]	≥ 1.3	
A-7	Market Exposure Time (the agent holds a non-zero position)	≥ 20%	
A-8	Stop-Loss Trigger Rate	≤ 50%	
B. Best Trained Agent Live Trading[17] Performance			
B-1	Excess Return vs. Buy & Hold	≥ 2 pp	Metrics are computed using the same evaluation logic as in backtesting, applied to realized live trading data. **Evaluation Period:** Minimum of one week.
B-2	Sharpe Ratio	≥ 0.3	
B-3	Sortino Ratio	≥ 0.4	
B-4	CAGR	≥ 5% annualized	
B-5	Maximum Drawdown	≤ 40%	
B-6	Profit Factor	≥ 1.1	
B-7	Market Exposure Time (the agent holds a non-zero position)	≥ 20%	
B-8	Stop-Loss Trigger Rate	≤ 60%	
C. Algorithm-to-User Performance Consistency			
C-1	Worst-Case User Return Deviation[18]	≤ 1 pp	User account returns are compared to algorithm-level returns. **Evaluation Period:** The exact same calendar period as used in Section B, to enable direct comparison.
C-2	Mean User Return Deviation	≤ 0.5 pp	
D. Server Performance and Reliability			
D-1	Tick Cycle Completion Within 5 Seconds Rate	≥ 95%	Measured from tick start until all relevant orders are submitted using system logs. **Evaluation Period:** Minimum of one week.
D-2	Tick Cycle Latency (Mean)	≤ 3.0 sec	
D-3	Multi-User Support Capacity	≥ 10 concurrent users	Evaluated under a simulated load of active sessions with no performance degradation. **Evaluation Period:** Minimum of one week.

[13] **Sharpe Ratio:** a risk-adjusted performance metric that measures the average excess return of a strategy relative to its return volatility. Higher values indicate better risk-adjusted returns.

[14] **Sortino Ratio:** a variation of the Sharpe Ratio that penalizes only downside volatility, focusing on harmful deviations from the target return rather than total return variability.

[15] **CAGR (Compound Annual Growth Rate):** the annualized rate of return derived from the total portfolio growth achieved over the evaluation period, normalized to enable comparison across different evaluation horizons.

[16] **Profit Factor:** the ratio between the total gross profit and the total gross loss generated by all trades executed during the evaluation period. Values greater than one indicate overall profitability.

[17] **Live Trading:** denotes real-time execution on a Kraken paper trading account, where strategies interact with live market data and exchange infrastructure without using real capital.

[18] **Return Deviation**: denotes the gap between algorithm-level returns and realized user-level returns, primarily caused by real-world execution effects such as latency, slippage, and order execution variability.

D-4	Strategy Parallel Execution	≥ 5 strategies executed concurrently	Verified by concurrent CPU core utilization during live operation. Parallelism is validated through operating system monitoring tools (e.g., task manager). **Evaluation Period:** Minimum of one week.
D-5	System Uptime Rate	≥ 95%	Calculated as (total runtime - downtime) / total runtime × 100 **Evaluation Period:** Minimum of one week.
D-6	API Failure Recovery Within 30 Seconds Rate	≥ 90%	Measured via system log analysis, verifying automatic recovery within 30 seconds of failure detection. **Evaluation Period:** Minimum of one week.
E. Data Quality and Completeness			
E-1	OHLCV Data Completeness Rate	≥ 99%	Measured by comparing the number of OHLCV candles successfully stored in the system to the number of candles expected for the configured timeframe. **Evaluation Period:** Minimum of one week.
E-2	VIX Data Completeness Rate (When Enabled)	≥ 90%	Measured by comparing the number of VIX data points successfully stored in the system to the number of data points expected for the configured timeframe (only in relevant VIX trading hours). **Evaluation Period:** Minimum of one week.
F. PineScript Conversion			
F-1	Conversion Success Rate	≥ 60%	Measured as the percentage of PineScript strategies that complete the full conversion and validation pipeline and pass all defined tests. **Evaluation Period:** Minimum of 10 conversion attempts.
F-2	Conversion Speed Per Strategy	≤ 10 min	Measured as the end-to-end time from PineScript input submission to successful completion of all conversion, validation, and testing stages. **Evaluation Period:** Minimum of 10 conversion attempts.
G. Software Quality			
G-1	Overall System Coverage	≥ 70%	Measured using pytest "coverage.py", reflecting executable code coverage by automated tests.
G-2	CI Pipeline Success Rate	= 100%	Measured as the percentage of GitHub Actions workflow runs that complete with a "success" status.
G-3	Dependency Security	= 0 vulnerabilities	Measured using automated dependency security scans integrated into the CI pipeline (e.g., GitHub Dependabot).

G-4	Documentation Coverage	= 100%	Measured using automated documentation coverage analysis tools (e.g., interrogate) to verify that all public modules, classes, and functions are documented with valid docstrings.
H. Client Application			
H-1	Application Startup Time (Login Screen)	≤ 3 sec	All client-side KPIs are measured using timestamped application logs to ensure accurate and reproducible latency measurements for startup, rendering, UI updates, and WebSocket events. **Evaluation Period:** Minimum of one week.
H-2	Dashboard Load Time (becomes fully interactive from successful user login)	≤ 3 sec	
H-3	Chart Rendering Performance (user interaction latency: zoom, pan, refresh)	≤ 500 milliseconds	
H-4	UI Update Latency (portfolio updates following trade execution)	≤ 1 sec	
H-5	WebSocket Stability Rate (uptime rate with automatic reconnection completed within 5 seconds following a disconnection)	≥ 99%	