Comprehensive RL Trading Agent Project Plan

Project Objective: "AlphaAgent"

To design, build, train, and rigorously backtest two distinct Reinforcement Learning agents:

1. **"TimingAgent" (Single-Asset):** A discrete-action agent designed to learn an optimal policy for timing entries and exits in a single, liquid asset, demonstrating core value-based RL methods.
2. **"PortfolioAgent" (Multi-Asset):** A continuous-action agent designed to learn an optimal policy for dynamic asset allocation across a defined universe of assets, demonstrating advanced policy-gradient and actor-critic methods.

This project will demonstrate key RL concepts: the exploration-exploitation tradeoff, reward engineering for risk-adjusted returns, state representation for complex market dynamics, and policy learning under non-stationarity.

Phase 1: Foundation - Environment, Data, & Tooling (Weeks 1-3)

This phase is the bedrock. The quality of the data and the realism of the simulation environment will directly determine the success or failure of the agents.

Step 1.1: Theoretical Foundation (MDP Formulation)

We frame trading as a **Markov Decision Process (MDP)**. This is crucial because, unlike supervised learning (which just predicts price), RL makes sequential decisions where each action (trade) affects the future state (portfolio) and cumulative rewards (PnL).

- **State (S):** A snapshot of all relevant information (market data, indicators, current portfolio holdings, cash).
- **Action (A):** The decision the agent makes (Buy, Sell, Hold, or specific portfolio weights).
- **Reward (R):** The feedback the agent receives after an action (e.g., change in portfolio value, risk-adjusted return).
- **Policy (π):** The agent's strategy, a function that maps states to actions ($\pi(A|S)$).

Step 1.2: Toolchain & Architecture Setup

- **Python Environment:** conda or venv.
- **Core Libraries:** pandas, numpy, scikit-learn, matplotlib, polars (for fast data I/O).
- **ML/RL Libraries:**
  - gymnasium (The modern OpenAI Gym standard for environments).

    - stable-baselines3 (Pre-built, SOTA algorithms like PPO, SAC, DQN).
    - PyTorch (The deep learning backend for the agent's neural networks).
- **Finance Libraries:** yfinance (data acquisition), pandas-ta (technical indicators), quantstats (performance analytics), exchange-calendars (handling market schedules).
- **Ops & Reproducibility:** mlflow or W&B (for logging runs, parameters, and model artifacts), YAML (for config-driven development).

Step 1.3: Data Layer & Acquisition

This layer is responsible for ingesting, cleaning, and aligning all data, ensuring **no look-ahead bias**.

1. **Define Universe:**
    - **TimingAgent:** A single, highly liquid asset (e.g., SPY ETF, or a continuous front-month future like ES).
    - **PortfolioAgent:** A small, diverse universe (e.g., [SPY, QQQ, GLD, TLT], representing Equities, Tech, Gold, and Bonds).
2. **Acquire Data:** Download daily or 15-min OHLCV (Open, High, Low, Close, Volume) data using yfinance or a dedicated provider.
3. **Obtain Benchmark:** Download data for a relevant benchmark (e.g., SPY).
4. **Handle Corporate Actions:** This is critical. All price series must be adjusted for **splits and dividends** to prevent false signals.

Step 1.4: Feature Store & Engineering

This module creates deterministic, **"as-of" timestamped** features. We do *not* feed raw prices to an agent. The state representation must be stationary and informative.

- **Price-based:** Log returns (np.log(price / prev_price)) over {1, 3, 5, 10 bars}.
- **Momentum Indicators:** RSI (14), MACD (12, 26, 9), Stochastic Oscillator (%K).
- **Trend Indicators:** 50-bar SMA, 200-bar SMA (and the crossover signal).
- **Volatility Indicators:** Bollinger Bands (%B, Width), ATR (Average True Range).
- **Volume/Microstructure (if intraday):** On-Balance Volume (OBV), Bid-Ask Spread, Volume Imbalance.
- **Portfolio Context:** Current position (e.g., 1 for long, -1 for short, 0 for flat), cash balance, time since last trade.

Step 1.5: Data Splitting (Critical for Validity)

To prevent overfitting and look-ahead bias, we must use a time-series-aware split.

1. **Simple Split (Baseline):**
    - **Training Set (e.g., 2000-2016):** The agent *only* learns on this data. (approx. 70%).
    - **Validation Set (e.g., 2017-2018):** Used for hyperparameter tuning. The agent *never* trains on this. (approx. 15%).
    - **Test Set (e.g., 2019-Present):** The "final exam." Held out until the very end for an unbiased performance assessment. (approx. 15%).

2. **Advanced Split (Plan 3 - Gold Standard):** Implement **Purged and Embargoed Walk-Forward Cross-Validation**. This simulates a realistic "retraining" schedule and is the standard for robust backtesting.

Step 1.6: Data Scaling

Neural networks require normalized inputs.

1. Initialize a StandardScaler or MinMaxScaler from scikit-learn.
2. **Fit the scaler ONLY on the Training Set:** scaler.fit(train_data).
3. Apply the *same fitted* scaler to the validation and test sets: val_data = scaler.transform(val_data). This prevents future information from "leaking" into the scaling parameters.

Phase 2: "TimingAgent" - Single-Asset Trader (Weeks 4-6)

Objective: Build and train a discrete-action (Long/Flat/Short) agent.

Step 2.1: Environment Design (TimingEnv)

Create a custom Python class TimingEnv(gym.Env) that simulates the market.

- **__init__(self, df):** Stores the preprocessed data.
- **State Space:** self.observation_space = spaces.Box(low=-inf, high=+inf, shape=(N_FEATURES + 1,)). This is the vector of all features *plus* the current position.
- **Action Space:** self.action_space = spaces.Discrete(3). This defines the agent's possible actions:
  - 0: Hold / Flat
  - 1: Buy / Go Long (if flat or short)
  - 2: Sell / Go Short (if flat or long)
  - *(From Plan 3):* We can map this to target positions: `{0: -1 (Short), 1: 0 (Flat), 2: +1 (Long)}*.
- **Internal State:** self.cash, self.shares_held, self.portfolio_value, self.current_step.

Step 2.2: Core Environment Logic (step function)

This is the engine of the MDP, defined as step(self, action):

1. **Execute Action:** Implement the logic for action 0, 1, 2.
2. **Simulate Fees (Critical):** Apply a penalty (e.g., 0.1% or 10bps) for every Buy or Sell to simulate **transaction costs** and **slippage (half-spread)**. This is non-negotiable for realistic results.
3. **Update Portfolio:** Calculate self.portfolio_value = self.cash + (self.shares_held * current_price).
4. **Calculate Reward (R):** This is the agent's feedback. This is a critical area for experimentation ("reward engineering").

    - **Baseline (PnL):** reward = self.portfolio_value - self.previous_portfolio_value. (Often too noisy).
    - **Risk-Adjusted (Sharpe Proxy):** reward = (portfolio_return - risk_free_rate) / portfolio_stdev. This intrinsically punishes volatility.
    - **Drawdown-Aware (Plan 3):** reward = PnL_t - (alpha * delta_Drawdown_t). This explicitly penalizes the agent for creating new, large drawdowns.
5. **Update State & Return:** Move to the next time step and return (new_state, reward, done, truncated, info).

Step 2.3: Agent Selection & Training (DQN)

For a discrete action space, **Deep Q-Network (DQN)** is the classic choice.

- **How it Works (Value-Based):** DQN uses a neural network to learn a "Q-function," $Q(s, a)$. This function estimates the *total future reward* the agent will get if it takes action a in state s and plays optimally thereafter.
- **Decision Making:** The agent simply calculates $Q(s, \text{Short})$, $Q(s, \text{Flat})$, and $Q(s, \text{Long})$ and picks the action with the highest Q-value.
- **Implementation:** model = DQN('MlpPolicy', train_env, ...).
- **Key Features (from Plan 1):**
    - **Experience Replay:** Stores (S, A, R, S') tuples in a buffer and samples mini-batches for training. This breaks time-correlation and improves sample efficiency.
    - **Target Network:** A second, slow-updating network used to create stable Q-value targets, preventing learning instability.
- **Advanced Variants (Plan 3):** Consider **Dueling DQN** (separates value and advantage) or **QR-DQN (Distributional RL)**, which learns the full *distribution* of returns, not just the mean, making it more risk-aware.

Phase 3: "PortfolioAgent" - Multi-Asset Allocator (Weeks 7-9)

Objective: Build and train a continuous-action agent that outputs portfolio weights.

Step 3.1: Environment Design (PortfolioEnv)

- **__init__(self, df):** Stores data for *all* assets.
- **State Space:** self.observation_space = spaces.Box(...). This is now a 2D matrix (or flattened 1D vector) of (N_Assets, N_Features) *plus* the current portfolio weights (N_Assets,).
- **Action Space:** self.action_space = spaces.Box(low=0, high=1, shape=(N_Assets + 1,)). This defines the agent's target allocation.
    - The vector represents weights for [Asset1, Asset2, ... AssetN, Cash].
    - This is a continuous, high-dimensional action space.
- **Internal State:** self.cash, self.shares_held = [s1, s2, ...], self.portfolio_value.

Step 3.2: Core Environment Logic (step function)

The step(self, action) function is now much more complex.

1. **Normalize Actions (Target Weights):** The agent's raw output (e.g., [0.8, 0.5, 0.2, 0.1]) won't sum to 1. We must **project** it onto a valid portfolio.
   - **Long-Only (Plan 3):** Apply a softmax function to the raw action vector to get target_weights that sum to 1.
   - **Long/Short (Plan 3):** Use a "budget projection" to enforce sum(|weights|) <= 1 (for 1x leverage) and sum(weights) = 0 (for dollar neutrality, if desired).
2. **Calculate Rebalancing:** Compare target_weights to current_weights. Calculate the *exact* trades (in shares) needed to rebalance.
3. **Simulate Fees (More Critical):** Apply transaction costs and slippage for *every single trade* executed during rebalancing. Add a **turnover penalty** (from Plan 3) to the reward function to discourage excessive, high-cost trading.
4. **Implement No-Trade Band (Plan 3):** If the *change* in a desired weight is below a small threshold (e.g., 0.5%), force no action for that asset. This prevents "churning" and saves on costs.
5. **Update Portfolio:** Calculate total self.portfolio_value.
6. **Calculate Reward (R):** Must be portfolio-level and risk-adjusted.
   - **Best Reward (Sharpe Ratio):** reward = (portfolio_return - risk_free_rate) / portfolio_stdev. This is the standard for portfolio management.
   - **Risk-Adjusted (Plan 3):** reward = portfolio_return - (lambda * portfolio_stdev). lambda is a hyperparameter for risk aversion.
   - **Advanced:** Add penalties for **factor exposures** (e.g., to reduce unwanted market beta) or for correlations during drawdowns.

Step 3.3: Agent Selection & Training (PPO / SAC)

For continuous action spaces, simple DQN doesn't work. We need **Actor-Critic** methods.

- **How it Works (Actor-Critic):** These methods use two neural networks:
  1. **The Actor:** This is the policy. It takes a *state* and outputs an *action* (the portfolio weights).
  2. **The Critic:** This is the value function. It takes a *state* and estimates the *total future reward* from that state (similar to DQN, but simpler).
- **Learning Loop:** The Actor takes an action. The Critic observes the result and tells the Actor, "That was a good (or bad) action," using a metric called the **Advantage**. The Actor then updates its policy to take "more good actions" and "fewer bad actions."
- **Algorithm Choices:**
  - **PPO (Proximal Policy Optimization):** The most common, stable, and robust choice. It "clips" policy updates to prevent them from changing too drastically, which is essential for stable learning in noisy financial markets.
  - **SAC (Soft Actor-Critic):** A highly sample-efficient off-policy algorithm. It uses **maximum entropy** to encourage exploration, which helps the agent discover new,

robust strategies.
    - **TD3 (Twin Delayed DDPG) (Plan 3):** Another SOTA algorithm for continuous control.
- **Implementation:** model = PPO('MlpPolicy', train_env, ...) or model = SAC('MlpPolicy', train_env, ...). These will require significantly more total_timesteps to train than DQN.

Phase 4: Backtesting & Evaluation (Weeks 10-12)

This is the "moment of truth." We must rigorously and honestly evaluate the final, trained agents on the **unseen Test Set**.

Step 4.1: Backtest Harness

1. Load your best-performing model (saved based on validation set performance).
2. Instantiate your environment using the **Test Set data**.
3. Write a simple loop that iterates through the test environment one step at a time.
4. At each step, get the agent's action using model.predict(obs, deterministic=True). deterministic=True is key, as the agent is now *exploiting* its learned policy, not exploring.
5. Log the info dictionary from env.step() to record portfolio value, trades, and costs.
6. **Crucial (Plan 3):** This backtester *must* be **event-driven** and **reuse the *exact* same cost, slippage, and execution logic from the environment**. This ensures an apples-to-apples comparison and is the "single source of truth."

Step 4.2: Define Strong Baselines

Your agent is only useful if it outperforms simple, no-effort strategies.

- **TimingAgent Benchmark:** "Buy and Hold" (B&H), and a simple "SMA(50/200) Crossover" strategy.
- **PortfolioAgent Benchmark:** "1/N" (Equal Weight) portfolio, and a "Hierarchical Risk Parity (HRP)" portfolio.

Step 4.3: Performance Analysis (quantstats)

Use quantstats.reports.html(agent_returns, benchmark_returns) to generate a full "tearsheet" report. Analyze these key metrics:

- **Performance:** Total Return, Annualized Return.
- **Risk-Adjusted (Most Important): Sharpe Ratio**, **Sortino Ratio** (penalizes downside volatility only), **Calmar Ratio** (return vs. drawdown).
- **Risk: Max Drawdown** (largest peak-to-trough loss), **Volatility**, **CVaR**.
- **Robustness (Plan 3):**
    - **Turnover:** How much did the agent trade? High turnover kills profits via costs.
    - **Factor Loadings:** Does the agent just have a 1.0 **Beta to market** (i.e., is it just a complex Buy & Hold)?
    - **Cost Sensitivity:** Rerun the backtest with 1.5x and 2x costs. Does the strategy still work?
    - **Ablation Studies:** Rerun training with key feature groups (e.g., volatility) removed.

How much did performance drop? This tells you what the agent *actually* learned.

Step 4.4: Final Review

Plot the agent's equity curve against the benchmark. Ask the hard questions:

1. Did the agent *significantly* outperform the benchmark on the *unseen test set*?
2. Is the outperformance (the "alpha") statistically significant, or could it be luck?
3. Does the agent's behavior make sense? (e.g., did the PortfolioAgent reduce equity exposure during the 2020 crash?).
4. Is the strategy robust to higher costs, or is the "edge" razor-thin?