ChatGPT

Step-by-Step Plan for an AI Trading System (ICICI Account, Teacher-Student & Multi-Agent AI)

1. Setup and ICICI API Integration

Step 1: Environment & Account Setup – Begin by setting up your development environment (e.g. Python with necessary libraries for ML, data, and APIs). Obtain credentials for the ICICI Direct **Breeze API**, which allows free access to extensive market data and trading functions [1]. Use the Breeze Connect SDK (available on PyPI) or REST endpoints to authenticate and connect to your ICICI demat account. This provides **up to 3 years of historical price data at 1-second granularity and live streaming OHLC data** for real-time feeds [1]. Ensure your ICICI account is enabled for API access and generate the required API keys (App Key, Secret, etc.).

Step 2: Cloud & Code Management – Plan to deploy the solution on the cloud for 24/7 operation. Use GitHub for version control. For free hosting, consider options like **GitHub Codespaces**, **Google Colab Pro (for scheduled runs)**, or a small cloud VM on a free tier. If persistent hosting is needed, you can explore community cloud platforms or an AWS/GCP free tier VM. Keep your repository updated on GitHub for easy collaboration and deployment. This cloud setup ensures the trading bot runs continuously without needing your local machine (aligning with *"cloud - github and free hosting"* from your requirements).

2. Data Collection and Frequency

Step 3: Historical Data Gathering – Leverage the Breeze API to download historical OHLCV data for your target stocks or indices. Aim for at least **1 year of historical data** (or more if available) to train your models [2]. Given that Breeze offers granular data, decide on the frequency that balances detail and noise for your strategy. Many algorithmic traders use **minute-level data for training and decisions** to capture intraday patterns [2]. For instance, one crypto trading bot trained on ~1 year of historical data and then operated on **1-minute intervals** for live decisions [2]. Using minute bars (or 5-minute bars) is often a good starting point for high-frequency strategies; it provides more data points and sensitivity to market moves, which can **increase model accuracy** by capturing short-term trends. However, if your strategy also needs a broader perspective, you might also collect daily data to derive longer-term trends – we will incorporate *both* intraday and daily signals in the model for completeness.

Step 4: Sentiment Data Collection – In parallel, set up a pipeline to gather sentiment data, since you indicated the need for a sentiment analysis module (similar to the crypto application). Identify relevant news or social media sources for the stocks in question: - **News Feeds:** Use financial news APIs (e.g. Alpha Vantage news, NewsAPI, etc.) or RSS feeds of popular finance news sites to get headlines and news articles about the companies or market.
- **Social Media:** If applicable, use Twitter API or forums (like Reddit) to fetch recent posts/tweets mentioning the stock or market sentiment.

Because sentiment can influence stock prices, collect this data at a reasonable frequency (e.g. daily or multiple times per day). Ensure timestamps of sentiment data are aligned with your price data for

merging later (for example, daily sentiment scores joined to daily market data or intraday sentiment if available).

Step 5: Live Data Stream Setup – Configure the Breeze API's **websocket/streaming** feature to receive real-time price updates for your assets [1]. This will be used in the live trading loop to make decisions on the fly. Also, set up a periodic job (or streaming if possible) to fetch new sentiment information in real-time (for example, pulling new tweets or news every few minutes). The live data feed from Breeze (which streams OHLC data in real time) ensures that a strategy tested on historical OHLC can be seamlessly applied live without changing data handling [3].

Step 6: Data Frequency & Update Strategy – Determine the model update frequency. For maximum accuracy, consider retraining or fine-tuning the model on new data periodically. For example, you might retrain weekly or monthly on the latest data, or even use online learning to update after each trading day. Using **higher-frequency data (minute-level)** will capture fine-grained patterns but also adds noise; ensure you apply smoothing or consider multi-timeframe features. In our plan, we will incorporate **both intraday and longer-term features** (addressing your "both" requirement) to let the model learn short-term patterns (from minute data) and overarching trends (from daily or weekly data).

3. Feature Engineering (Technical & Sentiment Inputs)

Step 7: Technical Analysis Indicators (TA) – Compute a range of technical indicators on the price data to provide the model with proven signals. Examples include: - **Trend indicators:** Moving Averages (MA/EMA), Moving Average Crossover signals, etc. - **Momentum oscillators:** Relative Strength Index (RSI), Moving Average Convergence Divergence (MACD). - **Volatility measures:** Bollinger Bands (for price range), Average True Range (ATR). - **Others as needed:** e.g. OBV (On-balance volume), Stochastic Oscillator, etc.

These indicators will serve as input features to the model or as separate expert outputs. For instance, a prior crypto bot project used **Bollinger Bands and RSI** among others [4]. We will similarly include such indicators which have predictive power for price movements. Ensure to compute indicators at appropriate timeframes (some on intraday data, some on daily data for trend context).

Step 8: Sentiment Analysis Module – Implement a sentiment analysis pipeline analogous to the one in the crypto application. This involves: - **Data Processing:** Clean and preprocess text from news headlines or tweets (remove noise, tokenize, etc.). - **Sentiment Scoring:** Use a pre-trained NLP model or library to classify sentiment as positive/negative/neutral. For finance-specific sentiment, a model like FinBERT or a simple polarity scorer can be used. Alternatively, utilize a cloud API for sentiment if available (keeping in mind cost-free solutions).
- **Feature Extraction:** Derive a sentiment score or indicator over a time window (e.g. average sentiment over last 1 day). This could be as simple as a daily sentiment index or as complex as an hourly sentiment time series if data is rich.

The sentiment module will output features such as "Market Sentiment Score" or "News Sentiment Trend" that the trading model can use. If implementing a multi-agent architecture, you might treat this as a **Sentiment Agent** that provides its own recommendation (more on multi-agent in a later step). According to the multi-agent trading framework example, a **Sentiment Analyst agent** would *"analyze social media and public sentiment… to gauge short-term market mood."* [5]. We will incorporate a similar capability to ensure the model isn't purely technical but also understands crowd sentiment.

Step 9: Macro/Fundamental Data (Optional) – Depending on your scope, you may also gather fundamental data (e.g. key financial metrics, earnings releases) or macro indicators (interest rates, economic data) if they impact your trading decisions. The crypto example included *Value at Risk and Macroeconomic data* as features [6]. For stocks, you could include things like market index trends or global cues as additional context. This step is optional but can enhance the model's robustness by providing a broader view of the market.

Step 10: Feature Consolidation: Combine the technical indicators, sentiment scores, and any other features into a unified dataset aligned by time. For each time step (e.g. each minute if intraday trading), your feature set might include latest OHLCV values, recent technical indicator values (RSI, etc.), and the latest sentiment score (perhaps the sentiment averaged over the last few hours or a daily score broadcast to all minutes of that day). Normalize or scale features as needed (e.g. RSI is already bounded 0-100, but prices may need scaling). This consolidated dataset will be used to train the AI models in the next steps.

4. Teacher-Student Learning Strategy

Step 11: Define Teacher-Student Framework – Design a training approach where a "Teacher" component guides the learning of a "Student" model. In our context, the teacher could be an algorithm or an existing model that sets up prediction tasks for the student and evaluates them: - The **Student model** is the one we ultimately deploy for trading – it could be a neural network that predicts price movement or price levels. - The **Teacher** can be conceptualized as either a larger reference model or simply a curriculum that presents challenges. For example, one can use a large pre-trained model or an ensemble as a teacher to generate ideal predictions, and train the student to mimic these (this is classic **knowledge distillation**). In fact, an advanced AI trading project demonstrated compressing a large model's trading reasoning into a smaller model via teacher-student distillation [7]. This involves *"knowledge transfer from a larger model to a smaller one"*, yielding a compact model optimized for trading decisions [7]. - Alternatively, implement the teacher as a **curriculum learning** mechanism: initially ask the student to perform simpler tasks (e.g. predict next-minute price on a small dataset), then progressively increase task complexity (e.g. predict further into the future or use larger datasets). This means the training starts with earlier historical periods and then moves forward in time.

Step 12: Iterative Training with Historical Tasks – Using historical data, simulate a training loop: 1. **Task Generation:** The "teacher" selects a point in time from history and asks the student model to predict the price at that point using data up to just before that point. For instance, "Given data up to Jan 2022, predict price on Feb 1, 2022." This is essentially creating a training sample where the ground truth (actual price on Feb 1, 2022) is known. 2. **Student Prediction:** The student model (initially untrained or pre-trained on generic data) makes a prediction. 3. **Evaluation (Reward/Punishment):** Compare the prediction to the actual historical price. Compute an error or reward signal (for example, reward could be `-abs(error)` or based on a trading profit if this was a trade). Provide this feedback to the training algorithm. 4. **Update:** Adjust the student model's weights to improve its performance on this task. This can be done via backpropagation (if framing as supervised learning on the true price) or via reinforcement learning update (if framing as an RL reward). 5. **Progressive Expansion:** Repeat the process moving forward in time – gradually include more recent data in the training. This way, the student is always tested on data it hasn't seen yet (avoiding data leakage), and it learns incrementally from *past to present*. Over time, the student is fine-tuned on the entire history up to the current date, which aligns with your idea of *"progressively increasing the data till the present – fine-tuning the student the entire time."*

This approach serves as a form of **online learning or walk-forward training**, ensuring the model adapts to changing market conditions gradually. It also mimics a teacher providing increasingly difficult challenges as the student improves (a curriculum). By the end of historical training, the student model should be well-tuned to current market behavior.

Step 13: Incorporate Reinforcement Learning (Optional Advanced Step) – To truly implement reward/punishment in an AI-agent sense, you can model the problem as a Reinforcement Learning environment. Using libraries like OpenAI Gym/Gymnasium, define a custom trading environment where: - **State:** includes recent prices, technical indicators, sentiment metrics at time *t*. - **Action:** the agent's decision, e.g. predict next price movement or decide trade action (buy/sell/hold). - **Reward:** based on the correctness of prediction or profit of a trade over the next interval. For example, a small negative reward for prediction error, or positive reward if a trade decision was profitable after some time.

The RL agent (our student) interacts with this simulated environment using historical data as the source of market moves. Frameworks like Gymnasium and stable-baselines3 can train a Deep Q-Network (DQN) or policy-gradient agent on the environment [8]. In fact, one trading agent project stacked reinforcement learning on top of supervised training, using a custom Gym environment and DQN to optimize strategies [8]. The environment (acting as teacher) "**simulates market conditions and rewards profitable strategies**," allowing the agent to learn from trial and error [9]. If time permits, integrating an RL fine-tuning phase after the initial supervised learning can further refine the model's decision-making by having it experience simulated trades and learn to maximize returns.

Step 14: Model Architecture (Student) – Choose an architecture for the student model that suits time-series forecasting: - A popular choice is an **LSTM or GRU network** (recurrent neural nets) which handle sequential data well, useful for predicting future prices from sequences of past prices and indicators. - Alternatively, consider a **transformer-based model** or temporal convolutional network if you have abundant data and want to capture long-term dependencies. - If formulating as classification (up/down moves), a simpler feed-forward network or even XGBoost model could be effective using engineered features. - Ensure the model input includes the features we engineered (technical indicators, sentiment, etc.), and output could be either a price prediction or a probability of price going up/down or even a suggested action.

Since we might integrate this model into a multi-agent framework, the output could be one "agent's opinion" (e.g. the technical model's forecast). Keep the model relatively small if you plan to run on free cloud resources (to ensure low latency and compliance with resource limits).

Step 15: Training Procedure – Train the model on historical data. If using the teacher-student curriculum approach: - Start with a subset (e.g. the first few months of data) to let the model capture basic patterns. - Gradually extend training data and continue training (or perform a rolling retraining where you train on a window, then slide the window forward). - Monitor training loss or reward over time to ensure the model is learning and not diverging. If using supervised learning, use a validation set (e.g. last 10% of historical data) to check for overfitting.

The *frequency* of model retraining can also be tuned here. A reasonable plan is to retrain or fine-tune the model with the latest data at some interval (say monthly) once in production, so it stays current. This frequency ensures maximum accuracy by incorporating recent market regime changes without overfitting to very short-term noise.

5. Multi-Agent Ensemble Approach

Step 16: Design Multi-Agent Architecture – To incorporate all aspects (technical, sentiment, etc.), organize the system into multiple **specialized agents or modules**, each focusing on a particular domain, and then combine their insights: - **Technical Analysis Agent:** Uses the technical indicator features and possibly a dedicated model to predict price movement or recommend trades based on technical signals (e.g. if RSI < 30 and upward momentum, recommend buy). - **Sentiment Analysis Agent:** Focuses solely on sentiment data. It could be rule-based (e.g. if sentiment is extremely negative, anticipate potential dip) or model-based (predict market reaction from sentiment trends). This agent provides a sentiment-derived prediction or risk level. - **(Optional) Fundamental/Macro Agent:** If fundamental data is included, an agent could analyze earnings reports, macro news, etc., though for short-term trading this might be less immediate. - **Risk Management Agent:** Monitors risk metrics (like volatility, Value-at-Risk as in the crypto bot [6]) and can advise to reduce exposure if risk is high. - **Trader/Decision Agent:** This agent's role is to aggregate the inputs from all other agents and make the final decision on whether to buy, sell, or hold at any given time. It can be a simple weighted logic or another model that takes the others' outputs as features.

For inspiration, note that the *TradingAgents* multi-LLM framework decomposed roles similarly: a **Sentiment Analyst agent** focused on public mood, a **Technical Analyst agent** focused on indicator signals, and a higher-level Trader agent combined their analyses [10] [11]. While our system may not use large language models for each role (to keep it lightweight and free), we mimic the spirit of specialization. Each module can be implemented as a Python class or process.

Step 17: Implement Agents/Models – Build out each agent: - Implement the **Technical Agent** logic using either the trained student model or even simpler heuristic rules for corroboration. For example, the technical agent might say "Bullish" if multiple indicators align bullishly, "Bearish" if many are negative, or defer to the model's numerical prediction. This agent could essentially *be* the trained ML model that looks at all technical features and outputs a forecast. - Implement the **Sentiment Agent** to output a sentiment score or recommendation. For instance, if average sentiment is strongly positive, the agent might flag a bullish bias, whereas strongly negative sentiment might warn of bearish conditions. You can set threshold rules on the sentiment score or train a simple classifier that maps sentiment features to a market move prediction. - Implement a **Combiner (Trader Agent)**: This can be as simple as a weighted average of the agents' outputs or a rule-based decision: - For example, if both Technical and Sentiment agents are confidently bullish and risk agent doesn't object, then issue a Buy signal. If they conflict (one bullish, one bearish), the trader agent might reduce position or stay flat. - Another approach is to train a second-level model (meta-model) that takes the outputs of technical model and sentiment analysis as inputs to predict the next move. This is effectively an **ensemble model**, which often improves accuracy [12]. In the crypto MAS bot, they *"trained ensemble models"* using multiple indicators including sentiment [12], and then used a **dynamic weighted model** to decide buy/sell [13]. We can follow a similar ensemble strategy: assign weights to the technical vs sentiment signals based on their recent performance. For instance, if the technical model has been more accurate recently, weight it higher, and adjust weights online with each tick's outcome (this is akin to online learning and was noted to create a robust adaptive algorithm [14]). - Implement the **Risk Management** as a simple check: e.g., don't allow more than X% of capital in one trade, or if volatility (ATR) is above a threshold, be more cautious on position size. This agent might simply output a scalar "risk factor" that can scale down the trade magnitude or veto a trade if risk is extreme.

Step 18: Agent Communication – Define how agents interact. This could be sequential (each agent writes to a shared state and the Trader reads all) or conversational. A simple implementation is: at each decision time, query each agent for its recommendation, then the Trader agent formulates the final action. You can log these for transparency (e.g., "Tech agent: BUY, Sentiment agent: HOLD, Risk:

medium → Trader decides BUY 50 shares"). If you want to emulate a "chat" between agents (like LLMs discussing), that's more complex and requires NLP synthesis of outputs – likely overkill for now. A straightforward polling of agent outputs should suffice, achieving the multi-agent approach in spirit.

6. Backtesting and Evaluation

Step 19: Backtest on Historical Data – Before deploying live, rigorously backtest the entire pipeline (the multi-agent system with the student model in the loop) on historical periods: - Simulate stepping through historical data (e.g. day by day or minute by minute) and at each step, feed the data to your agents, get a decision, and record what action would be taken (buy/sell/hold) and at what price. - Use the recorded actions to simulate the portfolio over time (assuming some initial capital). Track key metrics: total return, max drawdown, Sharpe ratio, win rate of trades, etc. - Compare the strategy's predictions against actual outcomes. Also evaluate the **model's predictive accuracy**: e.g., if it predicted price increase, how often did price actually increase? This can help tune thresholds or confidence levels required to act.

Step 20: Parameter Tuning – Use the backtest results to refine parameters: - Adjust technical indicator thresholds or periods if needed for better performance. - Tweak the sentiment scoring method or the weight it gets in decisions (maybe sentiment works better as a secondary confirmation rather than primary driver). - If the teacher-student training didn't yield desired accuracy, you may retrain the model with different network architectures or features. Check for overfitting or underfitting via validation set performance. - Also, evaluate different **frequencies**: maybe the model is more accurate on 15-minute intervals than 1-minute (due to noise). Test various bar durations in backtest to see where the accuracy or profit is maximized. The *"most suitable frequency"* might be the one where the strategy has the best Sharpe ratio or prediction accuracy in backtests – commonly 5-min or 15-min bars can sometimes outperform 1-min if 1-min is too noisy. Choose the frequency that gives a good balance of responsiveness and reliability.

Step 21: Full System Dry Run – Conduct a paper-trading simulation (if possible, using the Breeze API in a sandbox or with very small real orders disabled) for a few days. This "dry run" uses live data but doesn't execute real trades, logging all decisions. Ensure the system runs stable over extended periods and that the multi-agent logic functions as expected in real time. Adjust any performance issues (e.g., optimize code if data handling is slow, ensure API limits are not exceeded by spacing out data requests, etc.).

7. Deployment and Execution

Step 22: Deploy Trading Bot on Cloud – With the strategy refined, deploy the bot to a cloud environment for live trading. Since cost is a concern, utilize free resources: - If the strategy can run on schedule (e.g., only during market hours), you might use **GitHub Actions** with a cron workflow to trigger the bot at intervals (though continuous real-time trading via Actions is tricky due to run time limits). - A better approach is to use a cloud VM or container on services like **Render.com, Railway.app, or Oracle Cloud Free Tier**, which can run 24/7 at no cost or low cost. You will need to set up your repository code on that service and keep your API keys secure (use environment variables or a secure store). - Ensure the machine has the required Python environment and your code is launched at startup or via a scheduler. For example, deploy a Docker container that runs `python start_trading.py` which contains the main loop.

Step 23: Real-Time Trading Loop – In the deployed application, implement the live trading loop: - Connect to the Breeze API's live feed for your instruments. Subscribe to relevant tick or OHLC updates. -

At each new data point (e.g. each minute or each tick), update your feature calculations (update moving averages, recalc RSI, fetch latest sentiment if new info is available). - Run each agent/module to get their latest analysis. Then determine the trade action via the Trader agent logic. - If an action (buy/sell) is decided, use the Breeze API to **place the order** programmatically. The Breeze API supports live trading with functions to execute orders, which you can call with the decided quantity and price (market or limit). Make sure to include necessary safeguards: e.g., check available balance, avoid over-trading, etc., before sending an order. - Include a small delay or control loop to avoid excessive API calls (Breeze likely has rate limits). A one-minute frequency should be safe, but if using tick data, ensure your loop can handle many ticks per second – you might then aggregate or throttle decisions (e.g., only act at most once per minute).

Step 24: Logging and Error Handling – As part of the live loop, log every decision and action. Log files or a database should record timestamps, agent outputs (technical score, sentiment score), final decision, and order details. Also capture any API errors or exceptions (e.g., network issues, order rejections) and handle them (perhaps with retries or alerts). This logging is vital for later analysis and debugging.

Step 25: Security and Fail-safes – Because this connects to a brokerage account, implement safety checks: - If the bot or server crashes, ensure it can alert you or at least not leave positions unmanaged. A simple fail-safe is to place stop-loss orders with any buy, or have a separate script that can flatten all positions if the main bot is down. - Secure your API keys (don't hard-code them; use environment variables or encrypted storage). Never expose them in the GitHub repo. - Test the order execution on a small scale initially (micro trade amounts) to confirm everything works as expected in the live market.

8. Monitoring and Improvement

Step 26: Real-Time Monitoring Dashboard – Set up a dashboard or alerts to monitor the bot's performance in real time. This could be as simple as sending yourself messages (via email or Telegram API) when trades happen, or as elaborate as a live web dashboard: - The crypto project example used PowerBI updated each tick for monitoring [14] [15]. You can use Python libraries (like Plotly Dash or Streamlit) to create a small web app that displays current positions, P/L, recent signals, etc., and host it on the cloud (some platforms like Heroku or Streamlit Sharing can host such dashboards, though free tiers may have limitations). - At minimum, implement notifications for critical events: e.g. a trade executed, or if the portfolio crosses a certain loss threshold.

Step 27: Performance Review and Iteration – Continuously evaluate the live performance against expectations: - Compare live trading results to backtest results. If there's a large deviation, investigate if market regime changed or if there was an implementation bug. - Check if the multi-agent weighting is still optimal; you might find that the sentiment agent consistently adds value (or not). Re-weight or retrain agents as needed. Because our design allows **online weight updates** (as mentioned earlier where weights can be adjusted every tick based on performance [14]), you can automate some of this adaptation. For instance, if a recent batch of decisions were wrong, reduce confidence (weight) in that signal. - Monitor the **frequency** of trades and model inference. If using too high frequency data causes noisy decisions, you might decide to down-sample or only trade on more significant movements (avoid over-trading). The goal is maximum accuracy and profitability, not just activity.

Step 28: Scaling and Enhancements – Once stable, consider enhancements: - **Additional Agents:** You could add a **LLM-based analyst** for deeper analysis. For example, using an LLM to read news articles in real-time and provide a qualitative assessment (though this would require API access to an LLM like GPT-4, which might not be free). This could complement the sentiment score with more context (why

something is trending). - **Teacher-Student Upgrades:** If you find a very accurate but heavy model (maybe a large ensemble or an LLM agent) that's too slow for live use, use it offline to generate training data or guide the smaller student model further – a form of ongoing knowledge distillation. - **Reinforcement Learning in Live Trading:** The RL approach from training can be deployed to adapt in live trading as well. The agent can continue to learn from each trade's outcome (this is advanced and caution is needed to not drift the model in unpredictable ways, but it's possible to let the agent keep learning on new data each day). - **Multi-Asset and Portfolio Management:** Extend the system to handle multiple stocks or even other assets. This would involve spinning up agent instances per asset and a higher-level portfolio manager agent to allocate capital among them (similar to the portfolio manager role in the multi-agent framework [16]).

Step 29: Community & Updates – Stay active on the Breeze API community forums [17] for any updates or issues regarding the ICICI API. Also monitor the performance of your AI models regularly and retrain with new data as needed. Market conditions evolve, so periodic retraining (e.g. including the latest month of data) can keep the student model accurate. Document your results and refine the strategy over time, possibly incorporating more data or techniques as you gain confidence.

By following this comprehensive plan, you will implement a robust AI trading system that leverages your ICICI Direct demat account via API, uses a **teacher-student learning paradigm** for model training, and integrates a **multi-agent (ensemble) approach** combining technical analysis and sentiment analysis. This ensures all critical components – from data acquisition and technical indicators to sentiment signals and multi-agent decision-making – are covered in full, mirroring the successful elements of the reference crypto application. Good luck, and happy building!

Sources

- ICICI Direct Breeze API documentation – offers free historical (3-year, second-level) and streaming market data for algorithmic trading [1].
- Multi-agent crypto trading bot (MAS project) – demonstrated using 1-year of data and minute-by-minute live trading with an ensemble of **technical indicators and sentiment analysis**, combined via dynamic weighting for buy/sell decisions [18] [6].
- Chainstack Web3 AI Trading Agent – introduced advanced AI techniques (GANs, **teacher-student distillation**, reinforcement learning) for trading models [19], including compressing a large model's knowledge into a smaller model for trading decisions [7]. Also utilized Gymnasium and DQN for RL-based strategy optimization [8] [9].
- TradingAgents LLM Framework – an example of a multi-agent approach with specialized roles (Sentiment Analyst, Technical Analyst, etc.) collaborating to inform trading decisions [10] [20], inspiring our separation of concerns among agents.

---

[1] [3] [17] ICICIdirect - Online Share Trading in India at low brokerage
https://www.icicidirect.com/derivative/breezeapi

[2] [4] [6] [12] [13] [14] [15] [18] GitHub - astronights/A4-Trading: Automated Multi Agent Bot for Crypto Paper Trading
https://github.com/astronights/A4-Trading

[5] [10] [11] [16] [20] GitHub - TauricResearch/TradingAgents: TradingAgents: Multi-Agents LLM Financial Trading Framework
https://github.com/TauricResearch/TradingAgents

7 8 9 19 GitHub - chainstacklabs/web3-ai-trading-agent: Build an Autonomous Web3 AI Trading Agent (BASE + Uniswap V4 example)
https://github.com/chainstacklabs/web3-ai-trading-agent