ChatGPT

Modular End-to-End Research Trading Suite Design

Introduction

This document outlines a modular end-to-end trading research suite that integrates advanced AI agents and traditional quant methods into a cohesive pipeline. The goal is to support the entire trading workflow – from data ingestion and analysis to decision-making, execution, and evaluation – in a research-oriented setting. We emphasize a flexible, plug-and-play architecture where each component (LLMs, deep RL agents, etc.) can be developed and refined independently, then combined for robust trading strategy development. The primary market focus is U.S. equities, but the design is adaptable to other markets (e.g. crypto or international stocks). Throughout, we highlight the role of Large Language Models (LLMs) for insight generation, deep Reinforcement Learning (RL) for decision optimization, and a Hierarchical Reasoning Model (HRM) for high-level justification and scenario reasoning. We also discuss practical considerations like trading frequency, uncertainty estimation, and the roadmap from backtesting to live deployment.

LLM Agents for Sentiment Analysis and Alpha Factor Mining

Role of LLM Agents: Large Language Models will act as specialized research agents in our pipeline. They are used for tasks like **sentiment analysis** of news and social media, **alpha factor mining** (generating novel predictive features), and **hypothesis generation** for market behaviors. For example, an LLM can summarize the sentiment of daily news for each stock, propose new factor formulas based on financial theories or patterns in data, or generate hypotheses (in natural language or code) about why a certain strategy might work. Recent studies have shown that LLMs can incorporate domain knowledge and generate more resilient trading factors by capturing evolving patterns [1] [2]. Compared to traditional quantitative techniques, LLM agents add a layer of semantic understanding – they can parse unstructured data (news, reports, social posts) and produce structured insights or features for the model.

Model Choices – Proprietary vs Open-Source: We consider both proprietary LLMs (like OpenAI's GPT-3.5/GPT-4 APIs) and open-source models (like LLaMA-based FinGPT or Fin-R1) for deployment. Proprietary models often deliver state-of-the-art performance, but come with high usage costs, potential API latency, and limited transparency [3]. Open-source LLMs (with fine-tuning) offer cost-effective and customizable alternatives, though they may require more effort to reach comparable accuracy [4] [5]. For instance, **BloombergGPT** (50B+ parameters, trained on financial data) achieved strong results in financial NLP tasks, but its use in trading is constrained by extremely high compute costs and the closed nature of the model [6]. In contrast, **FinGPT** – an open-source financial LLM – has demonstrated competitive performance in financial sentiment analysis and is freely available [7]. Our design leans toward open-weight models that can be fine-tuned on in-house data for specialization, while still allowing integration of powerful proprietary models for certain tasks if needed (e.g. using GPT-4 for hypothesis generation on an as-needed basis).

Financial Domain Fine-Tuning: Domain-specific models tend to perform best on finance tasks. Prior examples include **FinBERT** (a BERT-based model fine-tuned on financial texts) which improved sentiment classification on financial news but was limited to batch/offline use [7]. More recently, researchers introduced **Fin-R1**, a 7B parameter LLaMA-based model tailored for financial reasoning. Fin-R1 uses a two-stage training (supervised fine-tuning on financial Q&A, followed by reinforcement learning-based training) and matches the performance of much larger models on reasoning tasks like FinQA and ConvFinQA [8]. This shows that smaller open models, when fine-tuned with financial data and possibly RL optimization, can achieve **comparable performance to proprietary LLMs** on specialized tasks [8] [5]. We will compare model families in terms of deployment trade-offs: for example, OpenAI GPT-4 may yield the best quality outputs but would involve sending data off-site and incurring latency/cost, whereas an in-house model like FinGPT or Fin-R1 can run locally with data privacy and be adapted over time, at the cost of some raw performance. Table 1 (below) summarizes some considerations:

- **Accuracy:** Proprietary models (GPT-4) typically lead, but fine-tuned open models (Llama2-based, etc.) are closing the gap [5]. Open models like Mistral-7B have even achieved parity with GPT-3.5 on some tasks when well-tuned [9].
- **Cost & Speed:** Open models run on our hardware have upfront compute costs but low incremental cost and can be optimized for speed. Some lightweight models can be very fast (e.g. Dolphin-2.6B achieving ~25ms per query) albeit with lower accuracy [10]. GPT-4 via API has higher latency (seconds per query) and token costs, making it less feasible for high-frequency use.
- **Transparency:** Open models allow inspecting weights and fine-tuning on proprietary data. This transparency aids debugging and compliance. Proprietary APIs are black-box – we only see outputs, not how the model arrived at them [3].
- **Maintenance:** An open model can be continuously improved (re-trained on new data or via reinforcement learning with feedback). Proprietary models improve only when the provider updates them, and fine-tuning may be limited or not possible.

Given these factors, the suite will initially incorporate **open-source financial LLMs** (such as FinGPT for sentiment and Fin-R1 for reasoning) as primary agents. FinGPT, for example, is specifically adapted to understand nuanced financial text and jargon, giving high accuracy on sentiment tasks [11] [12]. In tests, FinGPT outperformed other models like IBM-Granite and base LLaMA on financial sentiment, thanks to fine-tuning techniques (e.g. LoRA) that imbue it with domain-specific knowledge [13] [14]. These models ensure we maintain **domain relevance and high accuracy** while keeping the system self-contained. Proprietary LLMs can still be optionally integrated for one-off analyses (for instance, using GPT-4 to double-check a hypothesis or to generate a detailed natural language report), but they are not central to the automated pipeline to avoid dependency on external services.

Use Cases for LLM Agents: Within the pipeline, we will deploy multiple LLM-based agents with specialized roles:

- **News & Social Media Sentiment Agent:** An LLM analyzes news articles, earnings call transcripts, and social media feeds (like financial tweets or Reddit posts) to gauge sentiment. It outputs structured sentiment signals (e.g. a score from –1 to +1, plus key "reasons" or keywords driving sentiment). Using an LLM here allows understanding context and tone beyond simple keyword counting. For example, the agent can discern if a press release is positive ("exceeded expectations") or negative ("missed targets"), even with nuanced language. FinGPT is well-suited here; it was designed to handle real-time multi-source sentiment and provides both classification and a confidence logit for each sentiment decision [15] [16]. These sentiment signals become features for trading strategies (for instance, an extremely negative news sentiment might be a signal to reduce a long position).
- **Alpha Factor Mining Agent:** This is an exploratory agent that uses LLM capabilities to propose new predictive factors or formulas. Building on ideas from recent research (e.g. AlphaGPT or LLMFactor

approaches), the agent might take as input a description of known factors and market hypotheses, and generate novel factor definitions in pseudocode or mathematical form [1] [2]. It essentially uses its knowledge (both statistical patterns and financial theory embedded in training data) to suggest candidate alphas. For example, it could hypothesize a combined factor like *"net sentiment in last 3 days * trading volume surge"* or parse through regulatory filings to extract a complexity score as a factor. This agent accelerates the research process by **brainstorming new strategies** that human quants can then test. We will incorporate regularization or context so that these LLM-generated factors remain sensible – e.g. guiding the LLM with domain constraints (seeking factors related to momentum, valuation, etc.) to avoid nonsensical formulas [17] [18]. Notably, LLM-based factor generation can help counteract overfitting to historical data by introducing domain-informed creativity, as noted by AlphaAgent research [1].

- **Market Explanation & Hypothesis Agent:** An LLM that digests a combination of recent data (price moves, news events, economic releases) and produces a human-readable hypothesis for market behavior. For instance, if the portfolio had an unexpected loss, this agent might analyze news and say *"Market drop likely due to surprise rate hike talk – cyclical stocks particularly hit"*. While not directly driving trades, this module enhances interpretability and can generate new angles for investigation. It effectively serves as a **research assistant**, leveraging the LLM's broad knowledge (financial history, economic logic) to put current events in context. This output can feed into the HRM for scenario analysis or be presented to human researchers for qualitative insight.

In summary, LLM agents imbue the trading suite with **natural language understanding and creative hypothesis generation** capabilities that traditional quant models lack. They will be deployed in a modular fashion (each agent focusing on a specific input or task) and their outputs will be numeric factors or textual rationales that feed into the downstream decision modules. By combining both **proprietary-level intelligence** (via fine-tuned open models) and **custom domain prompts**, the suite can harness LLMs to continuously learn and adapt new signals in the ever-changing financial landscape.

Citations: The use of LLMs for sentiment analysis and factor mining is supported by recent research, showing improved performance when LLMs are tailored to financial data [7] [13]. Open-source financial LLMs (FinGPT, Fin-R1) have proven capable of matching larger models on finance tasks [8], offering a viable alternative to proprietary models with the benefit of lower cost and greater control [5].

Deep Reinforcement Learning for Portfolio Optimization and Execution

Role of Deep RL: Deep Reinforcement Learning techniques will be leveraged for two critical functions in the trading suite: (1) **Portfolio optimization** – dynamically adjusting asset allocations or positions to maximize returns for a given risk, and (2) **Trade execution decisions** – determining how and when to execute orders to achieve the best pricing and minimal market impact. Unlike supervised learning which predicts a target, RL learns a *policy* by interacting with an environment and maximizing a reward signal. In finance, this means an RL agent can directly learn to make trading decisions (actions) that optimize objectives like profit, Sharpe ratio, or drawdown, by trial-and-error on historical (or simulated) market data. By using deep neural networks as function approximators, **deep RL** can handle high-dimensional inputs (many features or assets) and learn complex non-linear strategies. Indeed, recent surveys highlight that deep RL has achieved impressive results on many trading benchmarks, often outperforming heuristic or traditional strategies under certain conditions [19].

RL Algorithms Considered: We will experiment with several state-of-the-art RL algorithms to find what works best across different trading horizons: - **Proximal Policy Optimization (PPO):** a popular on-policy algorithm known for stability in continuous action spaces. PPO could be used for portfolio rebalancing where the action is a vector of asset weight adjustments. PPO's clipped objective helps prevent unstable updates, which is important given the non-stationarity of markets. Researchers have used PPO variants successfully to discover trading policies that combine multiple alpha signals [20].
- **Soft Actor-Critic (SAC):** an off-policy, model-free algorithm that maximizes a trade-off between reward and entropy (encouraging exploration). SAC is well-suited if we treat trading as a continuous control problem (e.g. decide an allocation or an order size as a continuous percentage). Its ability to handle continuous action stochasticity can be valuable for **execution strategies** – for example, deciding what fraction of a large order to execute in each time slice. SAC's off-policy nature also allows leveraging past market data more efficiently (replaying experience), which is useful when market data is limited.
- **Decision Transformers:** a newer approach that reframes RL as a sequence modeling (Transformer) problem. Instead of learning a policy via value functions, a Decision Transformer takes the past states, actions, and returns and generates the next action as a sequence prediction task. This approach can take advantage of **trajectory data** and might better capture long-term dependencies (like multi-day trade sequences). In our suite, a Decision Transformer could be used for scenario-based planning, e.g., it could condition on a desired target return and generate a sequence of actions (trades) to attempt to reach that target. However, transformers are heavy models; we'd need to see if their latency is acceptable for our use case or if we only use them offline for strategy planning.

Each of these will be evaluated in simulation to see which yields the best outcomes for our objectives. It's possible we use PPO for the slower portfolio allocation decisions (due to its stability) and SAC or even DQN variants for faster execution tasks (where sample efficiency and off-policy training help).

Trading Frequency and RL Viability: One key consideration is how trading frequency (daily vs intraday) affects the suitability of deep RL and the latency tolerance for LLM involvement: - **Daily/Weekly Frequency:** At lower frequencies (daily or multi-day decisions), an RL agent has ample time to compute its policy and incorporate external inputs like LLM-generated signals. Here, latency is a minor issue – if an LLM sentiment agent takes a few seconds to run, it can still feed a daily trading decision. RL can be trained on daily data (which yields on the order of ~250 trading days/year – so a few thousand steps for a decade, possibly augmented by multi-stock experiences). Many academic works use daily data for RL portfolio management; though data is limited, the RL agent can augment it by considering many assets simultaneously (treating the combination as part of state) or by using resampling techniques. For weekly or monthly rebalancing, RL can certainly handle the long interval between decisions, but the limited number of training samples (few trades per year) means RL might not generalize well without incorporating prior knowledge. In those cases, supplementing RL with an LLM's insights or a model-driven bias can help (e.g., initialize the policy based on a known good strategy like mean-variance, then let RL fine-tune it).
- **Intraday Frequency:** For higher-frequency strategies (e.g. trading on 5-minute bars or even faster), pure deep RL (without LLM in loop) can be viable *if* it's optimized – RL policies once trained are just neural nets that compute actions quickly. In fact, there are instances where RL agents manage intraday trading or market making, operating on tick data with minimal latency (these are often custom environments with millions of training steps). However, incorporating an LLM agent at intraday timescales is challenging due to latency. A large LLM like GPT-4 might need several seconds per query, which is far too slow if we're trading every minute or second. Even smaller local models might introduce hundreds of milliseconds latency, which is an eternity in high-frequency trading. Therefore, for intraday strategies, our design limits how often the LLMs are invoked: LLM-based analysis might be done **pre-market or hourly** rather than on every tick. For

example, the LLM sentiment agent can update sentiment scores from news on an hourly basis, and an intraday RL agent can use those scores (which are held constant or updated infrequently) to trade on a minute-by-minute basis. The heavy computation is thus decoupled from the fast control loop. The RL agent (a lightweight neural network) can make split-second decisions based on the latest state (prices, technical indicators, last known sentiment score, etc.), ensuring we meet latency requirements. In summary, **LLMs are used off-line or at low frequency, whereas RL handles the high-frequency decisions** when needed. This arrangement acknowledges current latency limitations of LLMs [6].
- **Feasibility & Hybrid Approaches:** It's worth noting that not all trading problems are well-suited to RL. RL shines in **sequential decision problems** where intermediate feedback is available. A portfolio rebalancing or execution problem naturally fits this, as there's a sequence of trades and a final reward (e.g., episode return). But some strategies (like one-shot predictions of tomorrow's return) may be easier to tackle with supervised learning or statistical models. We will apply RL where it adds value – primarily in dynamic decision-making such as continuous portfolio optimization and adaptive execution. For one-step decisions (like "buy or not at today's close?"), we might instead use a simpler model (or treat it as a single-step MDP which reduces to a classification/regression).

Pros and Cons vs Traditional Methods: Deep RL offers a powerful, flexible framework but also comes with challenges, and it's important to compare it to traditional quant methods: - **Pros of Deep RL:** RL agents can learn directly from raw experience and optimize for *what we ultimately care about* (e.g., maximizing profit, risk-adjusted returns) rather than optimizing surrogate predictive metrics. They can theoretically discover complex trading strategies that a human might not anticipate, by exploring large action spaces. Notably, RL models combined with deep networks can handle high-dimensional inputs and discover non-linear patterns. For example, an RL agent could simultaneously consider dozens of indicators and learn an implicit market timing strategy – something very hard to design with manual rules. Another advantage is adaptability: an RL agent continuously learning (even paper trading) could adjust its policy as market dynamics shift. Additionally, RL doesn't require an explicit prediction model of the market's future returns; instead it directly learns what action to take, which sidesteps some difficulties of forecasting. Some studies have found that RL-based trading strategies can outperform traditional strategies on certain metrics or market conditions, often achieving better median returns with lower variability [19]. This is partly because RL can incorporate risk management into its reward function and learn to avoid large losses, which a standard predictive model might not do unless explicitly constrained.
- **Cons of Deep RL:** The downsides include **data inefficiency** – RL typically requires a lot of training episodes to converge to a good policy. Financial data is limited (we cannot run unlimited simulations of the real market beyond historical data), so RL agents risk overfitting the training period. Without careful regularization or training regimes, an RL agent might simply memorize a particular market regime from history and fail when conditions change. There's also the **exploration vs exploitation** issue: an RL agent might not have seen enough rare events (e.g., 2008 crash) during training to know how to handle them, whereas a human-designed model might explicitly account for such scenarios via stress tests. The performance of RL is sensitive to the reward design – if we set the wrong reward function (e.g., overemphasize short-term profit), the agent may learn pathological behavior that maximizes that reward but is undesirable (like taking extreme leverage to spike short-term returns). Designing a sensible reward that balances profitability and risk is non-trivial [21]. Moreover, RL training can be unstable; hyperparameters need tuning, and results can vary across runs. This stochasticity is problematic in finance where we usually prefer stable, robust strategies. By contrast, **traditional quant methods** (such as statistical arbitrage or factor models) are often more **interpretable and data-efficient**. A cointegration-based pairs trading strategy, for instance, uses economic theory to ensure two assets move together; it may perform reliably with limited data because it's grounded in a structural relationship. Similarly, factor models

like Fama-French use decades of economic intuition (value, momentum factors) and tend to be more stable – they won't dramatically change behavior unless the underlying data relationships shift significantly. They also provide clear rationale (e.g., "we hold value stocks because on average they outperform").
- **Best Use of Each:** In this suite, we plan to combine deep RL with traditional methods to get the best of both. Deep RL will be employed where **adaptive sequential decision-making** is required and where sufficient training data or good simulators exist (e.g., intraday trading strategies, multi-asset allocations). Traditional methods will complement or even override ML in areas where domain knowledge is strong. For example, if two stocks are known to be cointegrated due to arbitrage (like dual-listed shares or ETF vs NAV), a statistical arbitrage approach might be directly implemented for that pair, since it's well-understood and reliable. RL isn't needed to rediscover that relationship (and might overcomplicate it) – but RL could still be used to **enhance it** (for instance, an RL agent could learn optimal timing for trade entry/exit around the basic cointegration signal, or manage the position sizing in response to volatility). Another example: classical mean-variance optimization (MVO) gives an optimal portfolio for given forecasts and covariance. We might use MVO as a starting point, and then have an RL agent adjust the MVO output day-to-day based on short-term signals or to account for transaction costs – effectively a two-layer approach. In places where ML-based strategies struggle (due to non-stationarity or lack of data), we won't hesitate to use simpler methods. For instance, regime detection might be done with statistical methods (PCA or clustering on macro indicators) and then RL operates *within* a regime to fine-tune trades. The **complementary approach** ensures we leverage human financial insight (factors, cointegration, etc.) for baseline strategy and use RL for **incremental improvements and automation** that are hard to achieve by manual design.

Example – Execution via RL vs VWAP: A concrete scenario is trade execution. Traditionally, a large order might be executed via a VWAP schedule or TWAP (time-weighted) to minimize market impact. We can train a deep RL agent (perhaps using PPO or DQN) in a simulated market environment to outperform a VWAP benchmark – the agent can observe market liquidity and price momentum and decide when to trade faster or slower. RL can thus learn to undercut the VWAP if it detects price trending up (trade quicker) or wait if price is mean-reverting down, etc., which **outperforms static rules**. Such RL-based execution strategies have been shown to reduce execution cost in research settings, though they require a realistic simulator for training. We will incorporate Backtrader or custom execution simulators to train and test such an RL agent. If successful, this would be deployed to manage real order execution, with safeguards (e.g., not deviating too far from VWAP to avoid gaming).

In summary, deep RL adds a powerful toolkit to the trading suite, excelling in decision scenarios that are **dynamic, multi-step, and involve trade-offs**. Its integration will be modular: an RL agent is just another module that takes inputs (market state, signals) and outputs actions (trades). By comparing RL agents with traditional methods for each problem, we ensure that we only adopt RL where it provides a clear benefit. The combination of RL with LLM-derived insights is particularly exciting: RL can treat the outputs of LLM agents (like sentiment or fundamental scores) as part of its state input, thereby fusing statistical learning with semantic understanding. This could address some weaknesses of purely technical RL models – for instance, an RL policy might avoid buying a stock despite a price dip if the sentiment agent warns of bad news (thus avoiding a value trap). Our evaluation will explicitly benchmark deep RL strategies against simpler baselines to quantify the improvement and justify the added complexity [22] [23].

Citations: The advantages of RL in trading – ability to learn from interaction and handle complex, high-dimensional problems – are noted in literature [24]. However, challenges like large data requirements and sensitive reward design are also documented [25]. We will heed these lessons and combine RL with established quant techniques where appropriate.

Hierarchical Reasoning Model (HRM) for Decision Synthesis and Justification

What is HRM? The Hierarchical Reasoning Model (HRM) is a novel AI architecture introduced in 2025 that is inspired by the human brain's multi-level problem solving approach [26]. Unlike standard large language models that rely on a single sequence of transforms (often aided by chain-of-thought prompting), HRM features a two-tier recurrent structure: a **High-Level "Strategic Planner" module** and a **Low-Level "Rapid Executor" module**, working in tandem [26]. The high-level module operates at a coarse timescale – it formulates an abstract plan or reasoning outline. The low-level module works at a fine timescale – it executes detailed steps or computations required to fulfill each part of the plan. Crucially, the high-level planner does not move to the next subgoal until the low-level executor successfully completes the current task, ensuring a coordinated, stepwise reasoning process [27]. This design enables HRM to achieve deep, multi-step reasoning in a single forward pass through the network, without needing explicit intermediate supervision or massive chain-of-thought training data [28] [29].

Performance and Efficiency: Despite having a modest size (the reference implementation has only ~27 million parameters), HRM has demonstrated remarkable performance on complex reasoning tasks. It solved tasks like difficult Sudoku puzzles and maze pathfinding nearly perfectly, using only 1,000 training examples for fine-tuning [30]. Even more impressively, HRM outperformed models that are orders of magnitude larger (including some with extended context windows) on benchmarks like the Abstraction and Reasoning Corpus (ARC) [30]. In other words, HRM achieves a level of logical problem-solving that belies its small size, likely due to the way it structurally enforces a decomposition of problems. This is a huge advantage in our context: we can deploy an HRM for reasoning tasks without the heavy infrastructure needed for an LLM like GPT-4. HRM's **sample efficiency** and **computational efficiency** mean it can be trained and run with relatively low resources while still attaining high-quality reasoning. This is partially because HRM's architecture injects an inductive bias for reasoning, reducing the need for mountains of data or parameters to learn those patterns. For our trading suite, it means we can train an HRM on a limited set of scenarios (for example, historical market episodes or synthetic scenarios we create) and expect it to generalize well in reasoning about new situations.

Explainability and Structured Reasoning: A major selling point of HRM is improved **explainability** compared to black-box neural networks. Because HRM explicitly separates the *"what to do"* from the *"how to do it"* (planner vs executor), we can inspect the high-level plan to understand the rationale behind decisions. In fact, the internal state of the HRM's high-level module effectively represents a reasoning trace or outline. This is in contrast to a large monolithic transformer where the entire reasoning is entangled in billions of weights. In an HRM, one could query or log the sequence of high-level decisions made. For instance, an HRM used in trading might produce an internal plan like: *"Goal: justify buying AAPL. Step1: Check fundamental trend; Step2: Check sentiment; Step3: Check technicals; All signals agree, thus conclusion: Buy"*. The low-level module in this case would handle each step's details (e.g., retrieving the actual fundamental metrics, computing technical indicators, etc.), and the high-level ensures all needed steps are covered. Such a structured approach yields **inherent interpretability** – each sub-reason can be examined, and the final decision is supported by a logical progression. Early commentaries on HRM note that unlike traditional LLMs where attention weights hint at what was considered (but not *why* a decision was made), HRM *"exposes its internal states"* at each level for inspection, providing a more transparent view into the decision process (e.g., developers can see both the plan and the execution trace) [26]. This is extremely valuable in finance, where explainability is not just a nicety but often a regulatory requirement. If an AI suggests a trade, we ideally want to know the reasoning – HRM can offer that by design, whereas an LLM would need to be separately prompted to generate an explanation (which may or may not reflect its true internal reasoning).

Use in Our Pipeline: We plan to deploy the HRM at the **synthesis and evaluation stage** of the pipeline – essentially as a *"Chief Analyst"* that evaluates all signals and justifies final trade decisions. Concretely, after the specialized LLM agents produce their outputs (sentiment scores, fundamental analysis, technical signals) and the RL portfolio agent suggests an action, the HRM will take all this as input and perform a structured evaluation. It will answer questions like *"Does this trade make sense given the combined evidence? What scenario is this strategy assuming, and are there alternative scenarios?"* The output of HRM could be twofold: (1) a refined trading decision or recommendation (it might confirm the RL agent's action, or flag concerns), and (2) a **natural language justification or explanation** that can be logged or shown to human overseers. For example, the HRM might output: *"Buy AAPL – High-level reasoning: Planner noted positive fundamental trend and strong bullish sentiment. Executor confirmed EPS growth +10% (fundamental) and news sentiment +0.8. No contradictory signal found (technicals neutral). Thus HRM concludes buy is justified."* This kind of explanation increases trust in the automated system and helps the team audit the reasoning. If the HRM identifies a discrepancy (say sentiment is good but fundamentals are poor), it might output a more cautious recommendation (maybe "hold" or a smaller position, along with an explanation of the conflict). In this way, HRM acts as an **overseer agent** that brings all pieces together with rigorous logic.

Advantages over LLMs and Traditional Nets: Compared to using a single LLM for decision synthesis, HRM offers **more reliable reasoning**. Standard LLMs can certainly be prompted to analyze multiple factors and give a decision, but they often require careful prompt engineering and even then might produce plausible-sounding but ungrounded rationales. HRM's architecture inherently grounds the reasoning because the high-level plan depends on the low-level actually verifying sub-tasks. This reduces the risk of "illusory correctness" (LLMs sometimes explain a decision with reasons that *sound* good but weren't truly checked). Also, HRM's need for far less training data (no massive pretraining needed) avoids the **data leakage issue** – it won't have seen future data unless we give it, so it's easier to ensure its training doesn't include answers to its test scenario (unlike a pretrained LLM which might have training data overlap with test). Versus traditional neural nets (like a standard deep feed-forward network that directly maps inputs to an output), HRM provides structure and clarity. A traditional net might output a trade decision with 99% confidence but we have no idea if it did so because of valid signals or due to some artifact in data. HRM, by contrast, would have made intermediate assessments (like "signal X is strong, signal Y is weak") that we can inspect to validate the reasoning. In terms of performance, the initial HRM results suggest that a well-designed HRM can beat even large models on logically intensive tasks [31]. While financial markets have randomness that pure logic can't always solve, we expect HRM to excel in tasks like **scenario forecasting and consistency checks**. For example, HRM can be tasked to reason about *"if the Fed raises rates and inflation is high, what likely happens to tech stocks vs bank stocks?"* – a scenario question requiring chaining economic reasoning. A plain LLM might answer from training data (which could be outdated or not specific), but an HRM can simulate a multi-step deduction: (High-level: identify relevant relationships; low-level: recall that rate hike -> higher borrowing costs -> tech suffers, banks may benefit from margin; combine with inflation info etc.). Thus HRM can produce a more **robust and explainable forecast** of scenarios. We will use it to test strategy robustness: e.g., ask HRM to reason what could go wrong with a certain trade (it might enumerate potential adverse scenarios).

Recommended HRM Tasks: To leverage its strengths, we will assign HRM tasks that involve reasoning across multiple inputs and justifying conclusions: - **Cross-Checking Trade Decisions:** As described, HRM will

take the tentative trade signals from various agents and ensure they make sense together. It's effectively performing *sanity checks* and *conflict resolution*. This improves reliability – if one sub-agent (say momentum) gave an out-of-context signal, HRM can catch that before execution.
- **Scenario Analysis:** HRM can be run in a simulation mode where we feed hypothetical situations (e.g., "oil price up 10%, interest rates +0.5%") and let it reason how our portfolio would fare. This is akin to a stress test but done via logical inference rather than brute-force data. The output helps us see if the strategy has obvious vulnerabilities. HRM's architecture is suitable for scenario reasoning because it can break down the implications of each factor in the scenario step by step.
- **Trade Justification Reports:** For compliance or internal review, HRM can generate a short paragraph explaining each trade in human terms. This would be assembled from the reasoning trace. Such reports increase transparency: e.g., *"Trade: SELL XYZ. Justification: HRM observed that momentum turned negative and a recent news report was very bearish (sentiment -0.7). Fundamentals are neutral, but given the high valuation and negative news, the model finds the sell action prudent."* Having these justifications logged means even if the trade ends up losing, we have a rationale to examine. This is much better than a black-box trade with no context.
- **Improving Explainability for Users:** If this suite is used by a team of analysts, the HRM's outputs can help them trust and understand the system's behavior. It addresses the *"why did the AI do that?"* question in a concrete way. As noted in AI XAI literature, having an explanation improves user trust and model adoption [32]. We anticipate that HRM will be central to making the whole pipeline's decisions interpretable, something especially important if the strategies will eventually be deployed with real capital at stake.

In implementation terms, the HRM module will receive as input: a structured state containing key signals (sentiment summary, fundamental summary, technical indicators, current positions, etc.) and possibly the RL agent's proposed action. It will then produce an output that includes an action recommendation (which could agree or disagree with the RL agent) and a reasoning log. We might implement the HRM using the open-source reference or develop a simplified version with two recurrent networks ourselves. Since it's relatively small, we can even consider training or fine-tuning it on descriptions of historical market events (e.g., training the high-level planner on a set of known market cause-effect stories, and the low-level on executing simple financial calculations). Even without extensive training, the HRM can start with a prior (maybe even rule-based initially for the planner) and learn from experience as it runs (we can reinforce correct reasoning with human feedback or simulated feedback, effectively an RL fine-tuning of HRM's planner – analogous to how Fin-R1 used RL to improve reasoning [8]).

Comparison Summary: Traditional neural nets (like a standard deep policy network) are **opaque** and need tons of data; LLMs are powerful but also somewhat opaque and can be inconsistent in reasoning unless carefully prompted. HRM promises an approach that is **more data-efficient, consistent, and interpretable by design** [30] [26]. By deploying HRM in our suite, we aim to increase the reliability and transparency of the AI-driven decisions, which is crucial for adoption in a financial context.

Citations: HRM's architecture and capabilities are documented in recent work – it achieves complex reasoning tasks with a small model by splitting planning and execution [26]. Its high performance with limited data and outperformance of larger models on reasoning benchmarks are noted in the literature [30]. This supports our use of HRM for tasks requiring multi-step reasoning and explanation.

Modular Pipeline Architecture

Our trading suite is designed as a **modular pipeline**, with each stage handling a specific part of the process and feeding into the next. This modularity allows teams to develop, test, and upgrade components independently, and fosters clarity in the data flow (which is important for debugging and compliance). Below we describe the major components in order, from data ingestion to execution and evaluation. The pipeline is structured to be end-to-end: raw data comes in, gets transformed into useful features by various modules (including our LLM and analytical agents), then decision modules (like HRM or RL) produce trade actions, which are executed and finally evaluated. This corresponds to an idealized research workflow that can be eventually automated. Figure 1 illustrates the full system pipeline, and Figure 2 shows how the agent modules interact within this pipeline.

Data Ingestion

At the start of the pipeline is **data ingestion**, where we gather all necessary raw data from external sources. This includes:
- **Market Data (Price/Volume)**: We will pull historical and live pricing data for our universe of stocks (e.g., daily OHLCV bars or intraday tick data, depending on strategy frequency). APIs like Yahoo Finance, Alpha Vantage, or broker data feeds can be used for historical data. For live data, direct broker API (like Alpaca's market data API or Polygon.io) provides real-time quotes. The ingestion module will handle connecting to these APIs, scheduling data pulls (e.g., end-of-day batch for daily data, or streaming for intraday), and storing the raw data in a time-series database or in-memory structure.
- **Fundamental Data:** This includes financial statements (income statements, balance sheets, cash flow statements), earnings reports, analyst estimates, and macro-economic indicators. We may use providers like Financial Modeling Prep or Alpha Query for fundamentals, or even directly scrape SEC filings (EDGAR) for the latest 10-K/10-Q. Data ingestion will parse these sources – for example, retrieving quarterly fundamentals for all companies in the stock universe, or GDP/CPI numbers from government websites. Ensuring a unified format (e.g., a fundamental database keyed by date and ticker and metric) is part of this step.
- **News and Social Media:** We will ingest unstructured text data from financial news feeds (Reuters, Bloomberg news headlines, etc.) and social platforms (Twitter finance handles, Reddit investing threads). This might involve RSS feed readers or scraping news websites, as well as using Twitter's API or Reddit API to collect posts in near real-time. Because this data is high-volume and unstructured, we will apply some filtration at ingestion – e.g., focusing on news articles that mention stocks in our universe, or trending discussions on our watchlist of tickers. Each piece of text will be timestamped and tagged with relevant tickers or topics for downstream analysis.
- **Alternative Data (if any):** Optionally, we can ingest other data like sentiment from specialized sources (e.g., sentiment indices, fear/greed index), on-chain data for crypto if extended, or even satellite imagery data (if we ever incorporate such alt data). The architecture is open to plugging these in.

The ingestion module is designed to be **extendable and schedule-aware**. For instance, it knows that price data comes in continuously or daily, fundamentals quarterly, etc. It can be configured to run different jobs on different schedules. All ingested data is written to a centralized data store (which could be a set of CSV/Parquet files, a SQL database, or simply held in memory as Pandas dataframes during a backtest). The suite ensures heterogeneous data (structured numbers, text, etc.) are all gathered and time-synchronized as needed – one of the design goals like FinWorld had is *"native integration of heterogeneous financial data"* [22], which we adopt here by having a unified data pipeline.

Preprocessing

Once raw data is fetched, it goes through **preprocessing** to clean and structure it for analysis: - **Data Cleaning:** We handle missing values (e.g., fill forward stock prices for non-trading days or handle suspended trading days), remove obvious outliers or bad ticks, adjust prices for splits/dividends if needed to have continuous time series, and align different data sources to common timestamps. For text data, preprocessing includes removing HTML tags, boilerplate or duplicates, converting to lowercase (depending on the needs of the NLP model), and perhaps filtering irrelevant content. If using social media data, this might also involve language detection (to skip non-English if our models are English-only) or spam filtering.
- **Normalization:** We might normalize certain data fields – for example, scale fundamental ratios or standardize input features for RL. However, normalization has to be done carefully to avoid peeking into future data. All scaling or normalization in the research environment will be done in a rolling, out-of-sample manner. For example, technical indicators will be computed with only past data up to that time. If we standardize features (like z-score an indicator), we'll do it using a trailing window mean and std, not the full-sample statistics, to mimic live conditions. This avoids lookahead bias.
- **Feature Table Construction:** Preprocessing will merge various data into a tabular format keyed by time and asset. For instance, for each stock and each day, we may assemble a record containing that day's price change, volume, latest sentiment score (from LLM agent, computed previously), recent fundamental metrics (lagged), etc. This creates a **time-indexed feature matrix** that downstream modules like the RL agent can consume easily. We handle any necessary lagging – e.g., yesterday's news sentiment might be used to predict today's move, so we align accordingly. If our trading frequency is daily closing, we'll likely use T-1 features to predict T returns to avoid any forward bias. The preprocessing stage ensures this alignment and that no future data leaks into the present.

- **Dimensionality Reduction (if needed):** If we have an extremely large number of raw features (perhaps from many technical indicators or alternative data), preprocessing could also include a step to reduce feature count (through PCA or selecting top features by correlation or feature importance). However, initially we will likely hand-select a manageable set of features, so this might not be necessary.

The output of preprocessing is a **clean dataset** ready for analysis – this could be in-memory data frames during backtesting or a set of feature files on disk. By the end of this stage, we have structured numerical inputs and cleaned textual data prepared for the feature engineering and analysis stage. Maintaining this separation (raw vs preprocessed) allows us to quickly plug in new raw data sources with minimal changes to later stages (just need to add corresponding cleaning steps).

Feature Engineering and Embeddings

Feature engineering is where domain knowledge and creativity convert raw/preprocessed data into signals that might predict returns. This module produces **factors or embeddings** that represent the state of the market in a way our decision models can use: - **Technical Indicators:** We will compute standard technical features such as moving averages, RSI (Relative Strength Index), MACD (Moving Average Convergence Divergence), volatility measures (e.g., 20-day realized vol), momentum (returns over the past month, quarter, etc.), and mean-reversion indicators (like Bollinger band percent). These are formulaic transformations of price/volume data and are straightforward to compute with libraries or custom code. For each stock and each time, we'll attach these technical factor values. These serve as inputs to both ML models and as potential triggers in their own right.
- **Fundamental Factors:** Using the fundamental data ingested, we compute ratios and growth rates such as

P/E, P/B, ROE, earnings growth %, profit margins, etc. We might also include *factor momentum* – e.g., how a company's fundamentals have improved or deteriorated over the last year. These become part of a stock's feature vector. We ensure these are lagged appropriately (since fundamental data is published with delay). These fundamental features allow the strategy to differentiate companies on value and quality dimensions, complementing short-term technical signals.
- **Sentiment and NLP Features:** The outputs of our LLM sentiment analysis agent (and possibly other NLP analyses) are integrated here. For example, we take the sentiment scores that FinGPT or another model produced for each stock (from news and social data) [7] and use them as features. We might have multiple sentiment features: e.g., a news sentiment score (weighted average of news articles sentiment in last 24 hours), a social media sentiment score, and perhaps counts of positive vs negative headlines. If the LLM agent produces **embeddings** (vector representations) for news or company filings (embedding capturing semantic meaning of text), we could include those embeddings or some summarized form of them. For practicality, we may reduce an embedding (which might be 768-dimensional from a BERT model) to a few key dimensions via PCA or just take a sentiment score, as very high-dimension inputs can be hard to use in RL without a lot of training data. But these NLP-derived features are crucial for capturing information that isn't in the price – e.g., a highly negative news tone which could predict a price drop.
- **Alpha Factors & Custom Features:** Over time, as we (and our LLM factor-miner agent) develop custom alpha signals, we incorporate them here. For example, if the LLM suggests a novel factor formula (like a combination of growth and sentiment), we can evaluate it and, if promising, add it to the feature set. The pipeline might maintain a library of alpha factors that can be toggled on/off in experiments. Some factors could be simple (like *short interest ratio* if data is available, or *insider buying activity*), others could be synthetic (like principal components of a group of raw features). The modular design allows these to be added without affecting other parts – as long as the decision model is retrained or aware of new features.

- **Embeddings for HRM/LLM aggregator:** In addition to numeric factors, we may also generate **text or scenario embeddings** for use by the HRM or aggregator LLM. For example, we could embed the day's top news headlines via a language model and feed that embedding into the reasoning model so it has a sense of the day's context. The embedding condenses potentially a lot of info into a vector that the model can use. This is somewhat advanced and depends on model capability to use that vector; initially, we might stick to explicit features. But keeping this possibility means if an HRM variant can accept a vector of context (like "market regime embedding"), we can generate it here (maybe using an autoencoder or clustering on recent market data, then embedding cluster ID, etc.).

All these engineered features are appended to our dataset. The focus is on creating a rich representation of market state that spans technical, fundamental, and sentiment dimensions – akin to what a multi-strategy human team might consider. The pipeline supports storing both point-in-time feature values and possibly feature histories if needed (some models might want the recent trajectory of a feature; though RL can inherently use its recurrent or state mechanism to capture that, we might also explicitly include features like "3-day momentum" etc. to give short-term context).

Crucially, feature engineering is done **before** feeding data to decision models, and because it's modular, researchers can experiment with different feature sets by modifying this component and seeing how it affects performance, without touching ingestion or the models themselves. This separation of concerns makes the research process more efficient and less error-prone. It also means we can easily compare **ML strategies vs standard factor models**: e.g., we can run the pipeline where the final decision is just a linear combination of some engineered factors (a traditional factor model) and compare it to the RL policy using the same factors. The pipeline's modularity thus aids in benchmarking ML approaches against quant baselines [23].

Specialized Analysis Agents (LLMs and others)

This stage involves a collection of **specialized analysis agents** that provide deeper analysis on specific aspects of the data. We've already touched on these agents conceptually in earlier sections; here we integrate them into the pipeline: - **Sentiment Analysis LLM Agent:** This agent takes in news articles, social media posts, and possibly transcripts ingested for each stock (or the market as a whole) and produces sentiment scores and insights. Implementation-wise, this could be FinGPT or a similar model running in batch at set intervals (e.g., end of day for news, or every hour for intraday sentiment update). The agent might also output a short summary of the key news for each stock (for human use or to feed to HRM). The sentiment output is stored as features as described. It may also trigger alerts or qualitative flags (e.g., "XYZ had extremely unusual negative sentiment today") that could be used by HRM to reason about anomalies. According to prior research, using domain-specific LLMs like FinGPT ensures high accuracy in capturing financial sentiment nuances [33]. Our pipeline will maintain this as a separate module so that it could be swapped (for instance, if a new better model like a future "FinGPT-2" comes out, or if we extend to a multi-lingual sentiment model for international markets).
- **Fundamental Analysis Agent:** Potentially an LLM (or a smaller rule-based expert system) that analyzes fundamental data for each company. While we already have numeric fundamental features, this agent could provide a qualitative assessment. For example, it might read the management's discussion in a 10-K and classify it as positive/negative outlook, or detect if there are any red flags (e.g., an unusual spike in debt levels or a decline in cash that might signal problems). It might output textual analysis (like "Company ABC's revenue growth slowed and margin contracted – fundamental trend negative") and/or a score (fundamental outlook score). This is then used by HRM or as a feature. A model like Fin-R1 (financial reasoning LLM) could be fine-tuned for this purpose – answering questions about a company's health given its financial statements. Alternatively, we could use simpler rule-based logic (e.g., if earnings growth <0 and debt/equity > 2, flag fundamental weakness). As the pipeline evolves, a combination is possible: basic fundamental screens plus an LLM to interpret qualitative aspects (like tone of earnings call).
- **Technical/Momentum Agent:** This might not need an LLM since technical signals are numeric, but we might still implement a specialized agent that focuses on short-term price patterns. For instance, we could use a smaller ML model (like a convolutional network on price series or an LSTM that scans recent prices) to detect complex patterns (maybe candlestick patterns or microstructure anomalies) and output a signal (e.g., "intraday mean reversion signal: strong" or pattern classification like "head-and-shoulders detected"). This agent basically augments the traditional technical indicators with any pattern recognition beyond what simple indicators do. If such patterns are well-captured by our engineered features, this agent may be redundant. It's an optional component for research – we keep the design open to plugging in such an agent if needed.
- **Risk Analysis Agent:** Another optional agent could be one that continuously evaluates portfolio risk metrics (VaR, scenario risk) and outputs warnings or adjustments. This could be rule-based or an AI (like an agent that scans correlations and realizes portfolio is too concentrated). In a sense, this can be part of the evaluation module too, but having it as an agent means it can feed info back into decisions (e.g., "Risk agent says portfolio too concentrated in tech – HRM, consider that in decision").

Each specialized agent runs relatively independently on the data it needs, and produces outputs that are fed into the **Aggregation/Decision module** next. We orchestrate them such that their outputs are ready by the time a trading decision is to be made. For daily trading, that means all agents would produce their updated analysis after market close or before next open. For intraday, some (like sentiment) might update during the day, while others might run continuously (technical agent could produce signals every minute). A scheduler can coordinate these – e.g., run sentiment agent on new articles as they arrive, update the sentiment feature store, etc.

What's important is that the **agents are modular and specialized**: this improves performance (each can be optimized for its task) and debuggability (we can trace if, say, a sentiment error led to a bad trade). It also parallels the idea of **heterogeneous LLM agents in finance**, which has been shown to improve overall outcomes (e.g., a recent framework had multiple LLMs focusing on different data types to enhance sentiment prediction performance) [34] . Our design echoes that: rather than one model taking all data, we have purpose-built agents for different facets of the market.

Aggregation and Decision Module

This is the **brain of the entire pipeline**, where information from all analysis agents and features is consolidated to make a trading decision. We envision two layers here – an initial aggregation of signals, and then a final decision mechanism: - **Signal Aggregation (LLM/HRM):** We introduce an **Aggregation LLM/HRM module** that takes the outputs of all the specialized agents (and possibly raw features too) and synthesizes a coherent view. In the simplest form, this could be a weighted linear model or a rule-based logic (e.g., "if 2 out of 3 main signals (fundamental, sentiment, technical) are strongly buy, then aggregate signal is buy"). However, to fully utilize our AI approach, we lean toward using the **Hierarchical Reasoning Model (HRM)** or an LLM with a structured prompt to perform this synthesis. The aggregation module's job is to answer: *"Given all these indicators and analyses, what is our overall outlook for each asset, and what trades does that suggest?"*. For example, for each stock it might output a score like *buy +2, hold 0, sell –2* or a confidence in each action. The HRM described earlier is ideal here since it can explicitly reason through conflicts (maybe sentiment says buy but fundamentals say sell, etc.) and come to a justified conclusion [27] [35] . It can also produce an explanation with the conclusion, which we keep for logging. In cases where the reasoning is too complex to implement from scratch, we could start with a simpler **LLM-based aggregator agent**: e.g., a GPT-4 prompt that lists the key signals ("Sentiment is positive, Tech momentum is up, Fundamentals weak") and asks for a recommendation. But eventually, replacing or augmenting that with HRM will give us a more stable and explainable aggregator. The output of this stage is an **aggregated signal or preliminary decision** for each asset (or for the portfolio as a whole, depending on strategy). In a long-short portfolio context, this might be an intended weight or a ranking of assets by attractiveness.
- **Decision Logic / RL Agent:** After we have an aggregated view, the final decision of what trade to execute goes through the **Decision module**. There are a few design options here: it could be that the HRM/aggregator already effectively *is* the decision maker (for a simpler strategy, the HRM might just directly say "buy stock A, sell stock B"). Alternatively, and the approach we lean to for flexibility, the aggregator produces signals (like alpha scores or a recommended action per asset) which then feed into a **Portfolio RL agent**. This Portfolio RL agent takes those signals plus the current portfolio state and decides the actual trade actions (what to buy/sell and how much). The rationale is that the RL agent can learn optimal trading tactics (accounting for trade costs, capital constraints, etc.) while the aggregator focuses on analytical correctness of the signals. For instance, the aggregator might conclude "Stock A likely to rise, Stock B likely to fall" and assign scores, but the RL agent will decide *how* to allocate capital to those predictions (maybe going long A and short B in proportion, and considering existing positions or risk budgets). If the RL agent is not used, a simpler decision logic could allocate based on scores (like a rank-order weighting, or threshold rules). But incorporating RL here allows dynamic position sizing and sequential decision optimization. This agent effectively does the portfolio optimization: maximizing expected reward given the aggregated signals as part of its state. This is similar to approaches where an RL policy learns to translate model forecasts into trades, which can sometimes improve performance by learning to time entry/exit

better than static rules [36] [37]. In our pipeline, this Portfolio RL sits at the end of the chain: it observes features and aggregated signals (state), and outputs actions (trades). During backtesting, it would be trained (or evaluated if fixed) to ensure it integrates well with upstream signals.

- **Risk Controls and Overrides:** The decision module will also integrate any risk management rules before finalizing orders. For example, it will enforce limits like "no position > X% of portfolio", "sector exposures within Y bounds", "stop-loss triggers" etc. This can be done by the RL agent inherently (it could include penalties in reward for violating these), or by explicit checks after the RL outputs raw suggestions. We include this here to ensure the final output is always risk-compliant. For instance, if the RL agent said go 100% long in one stock, a post-check might scale that down if it breaches diversification rules. Alternatively, the HRM aggregator might foresee and adjust such extremes logically before passing to RL.

In essence, the Aggregation and Decision module forms a **two-step AI ensemble** – first collating information via reasoning (LLM/HRM), then deciding actions via learning (RL). In some implementations, these might blur into one (e.g., a single RL that incorporates reasoning steps internally), but we separate them for clarity and modularity. This separation also parallels human processes: an analyst (like HRM) forms a thesis and conviction, and a portfolio manager (like RL) decides the trade size and execution. We believe this yields better results than either alone, and each can be improved independently (e.g., improve HRM reasoning quality, or train RL with a new reward function) without fundamentally breaking the other.

The output of this stage is a **set of trade orders or target portfolio weights** to execute. For example, it might output: "Buy 500 shares of AAPL, Sell 300 shares of MSFT, hold other positions constant" or "Target weights: 10% AAPL, -5% (short) MSFT, etc." along with any contextual info (like the reasoning or confidence). This then moves to the execution component.

Figure 1 below depicts the flow from data and analysis modules through the aggregation and decision components, illustrating how signals converge into final trade decisions.

```
flowchart TD
    subgraph "Data & Ingestion"
        A[Market Data<br/>(Prices/Volume)]
        B[Fundamental Data<br/>(Financials/Indicators)]
        C[News & Web Data<br/>(News, Social Media)]
    end
    subgraph "Preprocessing"
        D[Clean & Align Data]
        E[Compute Technical Indicators]
        F[Structure Features Table]
    end
    subgraph "Feature Engineering"
        G[Technical Factors<br/>(e.g. MA, RSI)]
        H[Fundamental Factors<br/>(e.g. P/E, Growth)]
        I[Sentiment Features<br/>(from LLM analysis)]
        J[Custom Alpha Factors]
    end
    subgraph "Analysis Agents"
```

```
        style J fill:#eee,stroke:#aaa,stroke-width:1px
        K[**LLM Sentiment Agent**<br/>(analyze news/social)]
        L[**LLM Fundamental Agent**<br/>(analyze reports)]
        M[Technical Pattern Agent<br/>(detect trends)]
    end
    subgraph "Aggregation & Decision"
        N[**Aggregator (HRM/LLM)**<br/>Combine signals & reasoning]
        O[**Portfolio RL Agent**<br/>Optimize trades]
    end
    subgraph "Execution & Feedback"
        P[Execution Module<br/>(Broker API / Simulator)]
        Q[Evaluation & Metrics]
    end
    A --> D
    B --> D
    C --> D
    D --> E
    E --> F
    F --> G
    F --> H
    F --> J
    %% Sentiment features come from LLM agent analyzing News (C)
    C --> K
    K --> I
    F --> I
    G & H & I & J --> N
    L --> N
    M --> N
    N --> O
    O --> P
    P --> Q
    N --> Q
    F --> Q
```

Figure 1: End-to-End Pipeline Flow. *Raw data is ingested from various sources (market data, fundamentals, news/social). After cleaning and feature engineering, specialized analysis agents (including LLM-based ones for sentiment and fundamentals) provide insights. An Aggregator module (HRM or LLM-based) combines all signals and reasoning to form a cohesive view, which the Portfolio RL agent (or similar decision logic) uses to decide final trades. Trades are executed via a broker API or simulator, and results feed into an evaluation module for performance tracking.*

Execution Simulation and Live Trading

After a decision is made on what to trade, the **Execution module** takes over to carry it out. This part of the pipeline connects our strategy to the market (real or simulated) and handles the mechanics of order placement and fills: - **Backtesting Execution (Simulation):** In research mode, we will use a simulation engine to execute trades on historical data. We plan to use **Backtrader** (a popular open-source backtesting

framework in Python) for this purpose, given its flexibility in simulating different order types and tracking portfolio state. When the decision module outputs an order (e.g., “buy 100 shares of X at market”), Backtrader will simulate buying at the next bar’s open price (or whatever logic we configure) and update the portfolio holdings and cash. It will also simulate transaction costs (commission, slippage). This provides a realistic (though simplified) feedback to our strategy. Using Backtrader allows us to define custom brokers, custom slippage models, etc., to mirror real conditions as much as possible. The Execution module in backtest will operate in a loop with the strategy: for each time step, strategy provides orders, execution “fills” them and updates the portfolio, then the loop continues. This closed loop is essential for testing RL agents (as they need to get reward and new state after execution) and for evaluating how delayed fills or partial fills might affect strategy.
- **Paper Trading (Live Simulation):** When moving to forward testing on live data (but not real money), we’ll interface with broker APIs like **Alpaca’s paper trading API**. Alpaca provides a free paper trading environment where we can send orders (via REST API) and it executes them in a simulated manner on live market prices. Our Execution module will detect if we are in paper/live mode and instead of calling Backtrader, it will format the orders to Alpaca API calls (or IBKR paper trading, etc.). The module will then track the status of these orders (filled, pending) using the API’s responses. The benefit here is we test our strategy in real-time with live market data and realistic latency, without risking capital. It also forces our pipeline to handle real-world issues (network delays, data update timing, etc.). The Execution module encapsulates all this so that the rest of the pipeline (from ingestion to decision) doesn’t need to change between backtest and paper trading modes.
- **Live Trading:** Eventually, for deployment with real capital, the Execution module would connect to a real trading API endpoint (like Alpaca live trading or a broker’s execution service). Conceptually it’s the same as paper, just with a live account. Risk checks here become even more crucial – the module will double-check orders against limits and perhaps implement **circuit breakers** (e.g., if the strategy is unexpectedly sending a flurry of orders beyond normal, pause and alert humans). For safety, the Execution component could require a human confirmation for certain actions in early phases of live trading (like trade size above a threshold). As confidence grows, it can be fully automated. The modular design with the Execution component separate means shifting from simulation to live is mostly a configuration change.

- **Order Types and Smart Execution:** The Execution module will support various order types (market, limit, stop) as needed. For backtesting, market orders are easiest to simulate (assuming we get the next bar’s price). For live, we might prefer limit orders to control entry price. The module can incorporate logic for order slicing (if we need to execute a large order, it could break it into smaller chunks to minimize impact – this logic could also be guided by an RL execution policy as mentioned earlier). We can also integrate **smart order routing** or liquidity-seeking algorithms here if using a broker that supports it. In the initial version, we likely keep it simple: send a market order or limit near current price, assuming our strategies aren’t so high frequency that this matters drastically. But if intraday strategies are pursued, we will implement at least basic **execution optimization** (like VWAP slicing or RL-based slicing).

- **Portfolio Accounting:** The Execution module (along with Backtrader or broker) maintains the portfolio state – how many shares of each stock we hold, current cash, margin usage, P&L, etc. This state is fed back into the pipeline. For example, the RL agent’s state includes current positions, and the HRM reasoning might consider current exposure (“we are already heavily long tech, perhaps hold off adding more”). Thus, the pipeline should query the Execution/Portfolio state as part of the input for each decision cycle. In backtest, Backtrader provides this via its broker model; in live, we can keep track of it internally or query the broker.

The Execution stage is where the “rubber meets the road” – all the analysis culminates in real trades. A robust execution module ensures that slippage and transaction costs are accounted for, and that orders are executed in a manner that doesn’t deviate too much from our model assumptions. For example, if our strategy is sensitive to fill prices, we may simulate pessimistic slippage in backtesting to avoid overly optimistic results [38]. We will incorporate that in Backtrader (e.g., assume we get the worst price of the bar for a market order, or a certain $0.01/share slippage). These settings can be refined by comparing paper trading outcomes to backtest predictions and adjusting.

Evaluation and Verification

Once trades are executed, the pipeline closes the loop with an **evaluation and verification** module. This component is critical for research to understand performance, diagnose issues (like biases or leaks), and ensure the strategy is behaving as expected: - **Performance Metrics:** The evaluation module computes a variety of metrics on the strategy’s performance. Key metrics include cumulative returns, annualized return, annualized volatility, Sharpe ratio, maximum drawdown, Win/Loss ratio, profit factor, exposure by asset or sector, etc. These are calculated for the backtest period or ongoing for live. We’ll also compare performance to relevant benchmarks (e.g., S&P 500 index for an equity strategy). For example, it will compute *alpha* and *beta* relative to the S&P if appropriate, the information ratio against the benchmark, and so on. In a multi-strategy scenario, it might track performance attribution (how much each component contributed).
- **Bias and Error Checking:** The verification part will specifically look for signs of **memory bias or data leakage**. One major concern is that our LLM agents or models could be inadvertently using information they shouldn’t have at a given time. For instance, as discussed earlier, a pretrained LLM might “remember” a news outcome from the past. The evaluation module will run tests to detect this: for example, we can do a backtest where we shuffle news dates (as a stress test) and see if the strategy oddly still predicts events too well – that would indicate lookahead. Another approach is to hold out a period that we know the LLM was trained on and see if performance is suspiciously good there compared to genuinely unseen periods [39]. If we detect such issues, we adjust the pipeline (maybe limit LLM’s knowledge or use an earlier cutoff in prompts). The evaluation module also monitors trading behavior for **overfitting signs**: e.g., if one particular rare event in history contributed most of the strategy profits, that’s a red flag of potential overfit. We can simulate slight variations of the historical data (jackknife or bootstrap the time series) to see if performance is stable – a technique akin to *Monte Carlo simulation* on price data for robustness.
- **Memory Bias Mitigation:** In iterative strategy development, there is also the danger of “researcher bias” – the team might unconsciously tune the strategy to past data. To mitigate that, our evaluation module enforces a strict train/validation/test separation for any model training (like RL or supervised models). We use one period for training RL, and a distinct period for backtest evaluation. Additionally, we plan to incorporate *walk-forward testing*: periodically re-train models on expanding windows and test on the next slice, to simulate how it would perform as time moves on. This can highlight if a model is consistently good or just lucky in one period. The evaluation output includes these results for transparency.
- **Benchmark Comparison and Statistical Tests:** We will compare the strategy to simple benchmarks (like buy-and-hold, or a moving average crossover system, etc.). The evaluation will report not just raw metrics but also whether the outperformance is statistically significant (e.g., t-tests on excess returns, p-values, or a reality check test for data mining bias). This helps ensure we’re not fooling ourselves with random good luck. If the strategy doesn’t significantly beat a basic benchmark after accounting for variance, we know more work is needed.
- **Live vs Backtest Divergence:** When we transition to paper trading, the evaluation module will closely track the **live performance vs the backtested expectation**. It will log any material deviations. For example, maybe in backtest the average daily return was +0.1% with 1% volatility, but in live paper the first

month is -2% with higher volatility – the evaluation would flag this and trigger analysis into why (maybe regime change, maybe an implementation issue). This feedback is crucial to validate that our simulated results actually hold in forward testing. Often differences can come from execution slippage, which the evaluation can quantify by comparing simulated fill prices to actual paper trading fill prices, etc.
- **Model Monitoring:** If our pipeline uses online learning (continually updating models) or has any adaptive components, the evaluation will monitor those as well – e.g., monitor the LLM agent's sentiment accuracy over time (perhaps using known sentiment benchmarks or market reaction), or monitor the RL agent's exploration vs exploitation (ensuring it's not diverging). Essentially, treat the models themselves as needing evaluation. For the LLM agents, we can periodically evaluate their outputs on a labeled dataset (say, check sentiment agent against a known sentiment classification dataset) to ensure they remain calibrated. For RL, we monitor things like average reward, and whether its action distribution shifts unexpectedly.

All evaluation results are recorded and typically visualized in reports (could be in Obsidian itself or exported to dashboards). We expect to generate an **evaluation report notebook** for each backtest or live run, which includes equity curves, drawdown plots, bar charts of factor performance, confusion matrix for signals vs actual outcomes, etc. This forms the basis for iterative improvement.

Importantly, the evaluation module helps guard against **overfitting and false strategies** – a pitfall especially when using powerful models like LLMs, which, if not careful, might leak future information. We explicitly address the data leakage issue highlighted in recent research [40]: if an LLM was trained on historical data that overlaps our backtest period, traditional backtesting becomes invalid because the model had knowledge of future events [39]. Our evaluation strategy to mitigate this is **forward-looking testing**: we structure tests such that the LLM's knowledge cutoff is always before the test period [41]. For example, if we use GPT-4 with knowledge up to 2021, we avoid evaluating its strategies on 2019-2020 data where it might have seen outcomes; instead, test from 2022 onward (or use an LLM that knows nothing beyond certain date). If we fine-tune an LLM on past data, we'll use rolling origin evaluation – not test on the same data it was fine-tuned on. The evaluation module automates some of this by having configuration to exclude periods known to be in training data. Ultimately, the true test will be paper trading in a **live-forward manner**, which the evaluation module facilitates by collecting live results and comparing them to expectations. The roadmap section will detail how we transition from backtest to live.

Finally, evaluation is not one-time: it runs continuously in live trading, monitoring performance and raising alerts if something is off (for example: a alert if drawdown exceeds a threshold or if the strategy's characteristics deviate from historical norms). This ties into risk management – we might have an automatic cutoff to stop trading if certain eval metrics breach limits.

With all components described, the pipeline forms a closed feedback loop: data -> features -> signals -> decisions -> execution -> feedback -> (back to analysis for improvements). This design mirrors professional trading system workflows and ensures that we can research and deploy strategies in a systematic, controlled, and **scientifically rigorous** manner.

Agent/Module Interconnectivity Diagram

To visualize how the various agents and modules discussed above interact within the system, consider the following diagram (Figure 2). It highlights the flow of information between specialized agents, the aggregator/HRM, the RL decision agent, and the execution loop:

```
flowchart LR
    subgraph "Market Environment"
      X[(Market Data<br/>(Live or Simulated))]
    end
    subgraph "LLM Agents"
      A1-->|news & social|SentimentLLM[Sentiment LLM Agent]
      A2-->|fundamentals|FundamentalsLLM[Fundamental LLM Agent]
      A3-->|price data|TechAgent[Technical Analysis Agent]
    end
    subgraph "Aggregator & Reasoner"
      Aggregator[**HRM/LLM Aggregator**]
    end
    subgraph "Decision Maker"
      RLAgent[**Portfolio RL Agent**]
    end
    subgraph "Execution"
      Exec[Execution Engine<br/>(Broker/Simulator)]
    end
    subgraph "Evaluation"
      Eval[Evaluation & Metrics]
    end
    X -- data feeds --> SentimentLLM
    X -- data feeds --> FundamentalsLLM
    X -- data feeds --> TechAgent
    SentimentLLM -- sentiment signal --> Aggregator
    FundamentalsLLM -- fundamental insights --> Aggregator
    TechAgent -- technical signals --> Aggregator
    Aggregator -- trade rationale & suggested action --> RLAgent
    RLAgent -- orders (actions) --> Exec
    Exec -- fills/portfolio state --> RLAgent
    Exec -- trade outcomes --> Eval
    Aggregator -- reasoning & signals --> Eval
    X -- market returns --> Eval
```

Figure 2: Agent Module Interactions. *Specialized LLM agents (for sentiment, fundamentals) and a technical analysis agent process raw market inputs and produce signals. These feed into an Aggregator (HRM or LLM-based reasoning module) that synthesizes a trading recommendation with justification. The Portfolio RL Agent then decides specific trade orders which go to the Execution engine (connected to a broker or simulator). The execution results (fills, new positions) are fed back to the RL agent (closing the loop for state update) and to the Evaluation module. The Evaluation module also receives the aggregator's rationale and actual market returns to assess performance and consistency.*

This diagram shows at a high level how each component is both **modular** and **interconnected**. The LLM agents don't directly talk to each other but their outputs meet in the Aggregator, ensuring each does its focused task. The RL decision agent is downstream, using all aggregated info to take actions, and it interacts with the Execution environment in a feedback loop (critical for RL learning and for reflecting current positions). Finally, evaluation observes multiple points (market data, agent outputs, executed trades) to give a comprehensive assessment.

Stock Universe and Markets

The trading suite is primarily configured for **U.S. equities** as the main universe, but we design it to be flexible so that with minimal changes it can support other markets like cryptocurrencies or international stocks (e.g., Chinese A-shares). Here's how we handle universe and market considerations:

Universe Selection (U.S. Equities): We will typically focus on a predefined set of stocks – for example, the S&P 500 constituents or a similar large-cap universe. Using S&P 500 (or Russell 1000) ensures we have relatively liquid stocks with abundant data (prices, news, analyst coverage) and reduces issues with illiquidity or extreme volatility that smaller stocks might have. The universe selection can be implemented as a module or configuration that feeds into the data ingestion stage (telling it which tickers to pull). In practice, we might start with a smaller universe (say top 50 stocks by market cap) for initial development to reduce data load and then scale up. The suite can also allow **dynamic universes** – for instance, one could use a filter (like "top 100 stocks by momentum" or "tech sector stocks") that updates periodically. QuantConnect and other platforms use the concept of Universe Selection models [42] , and similarly we can incorporate logic to adjust our stock list (with care to avoid lookahead bias – any universe change rules must use only prior information). Initially though, a stable universe (like all S&P 500 members as of a start date, rebalancing only when indices rebalance) is simplest and avoids survivorship bias if we include historical members properly.

Liquidity and Data Availability: Emphasizing U.S. equities means we have high-quality, relatively clean data and fewer trading restrictions (no daily price limits as in some markets, etc.). We will still be mindful of corporate actions (splits, mergers) in our data handling. Given that our strategies might involve shorter-term trading, we probably will exclude the very lowest liquidity names even within S&P 500 (though all S&P 500 are liquid). If we expand to, say, Russell 2000, we'd need to be more cautious with execution assumptions, as some small caps can have wide spreads. The suite can incorporate liquidity filters (e.g., average daily volume > X) in the universe definition if needed.

Extensibility to Other Markets: - *Crypto:* If we want to apply the suite to cryptocurrency markets, we'd mainly swap out the data sources (using crypto exchange APIs for price, crypto-specific news sources, on-chain data ingestion, etc.). The agents and pipeline largely remain the same – e.g., sentiment analysis agent could analyze crypto Reddit or Twitter (maybe requiring tuning the LLM to crypto slang), technical features work similarly (though crypto trades 24/7, so our system must handle non-stop data). Execution for crypto would connect to an exchange API (like Binance or Coinbase Pro). Latency is more of a factor if doing intraday crypto, but again, if we limit LLM calls frequency, it's manageable. One consideration is that crypto has different market structure (no closing time, more volatile, often higher retail participation). Our RL agent's reward function might need adjusting (volatility targeting, etc., for crypto's nature). But overall, the modular design means to switch to crypto, we just "plug in" a crypto data module and possibly retrain/tune agents on crypto data. The flexibility was inspired by platforms like FinWorld that support multiple stock pools/markets in one framework [43] .
- *International Equities (e.g., China):* For markets like China A-shares, we'd have to handle multi-language news (Chinese news sentiment – which means our sentiment LLM needs to understand Chinese or we use a separate Chinese LLM). We also adjust trading calendar (Chinese market has different trading hours, holidays). Universe selection might be something like the CSI 300 index (top 300 Chinese stocks). The

pipeline could incorporate a translation step in ingestion (e.g., translate Chinese news to English for our English-trained LLM, or better, use a Chinese LLM). With modern models like multilingual transformers, sentiment analysis can be extended to Chinese. Fundamental data for Chinese companies might be less accessible or in different formats, but again, the ingestion module can be adapted. One difference: Chinese markets sometimes have daily price move limits (e.g., ±10%), which our simulation should handle in execution (Backtrader can simulate hitting a limit – orders not filling if limit reached). Also, data quality might be a bit more challenging with different standards. But none of these are architectural blockers – it just means swapping or extending some submodules. By keeping those concerns localized (e.g., separate config for market = "China" vs "US"), the core reasoning and decision modules remain unchanged.

Universe Flexibility: We plan to maintain the universe definition in a config file or database table, making it easy to change. Researchers can easily run the pipeline on, say, *"DJIA 30 stocks"* by providing that list, or *"Top 10 crypto coins"* etc. The ingestion module will fetch for whatever tickers list is given. The pipeline doesn't hardcode any particular asset names. This flexibility is helpful in research – one might test on a small universe for speed, then gradually scale up. It also allows running multiple instances of the pipeline on different universes in parallel (for example, one could deploy one model on US equities, another on European equities, etc., sharing much of the codebase but separate configs).

Sector and Category Handling: Within a broad universe like S&P 500, the suite can also pay attention to sub-universes (sectors, industries). For instance, the HRM might reason at a sector level ("tech sentiment is down overall"), or the RL agent might have state features aggregated by sector. We can compute sector-level factors (like average momentum of sector) in feature engineering if needed. Universe selection module might allow focusing on a sector for specialized strategies. Again, modular design lets us incorporate this without redesign – just additional features or constraints (like "don't overweight any one sector > X%").

Data Coverage and Survivorship: One practical note: to avoid survivorship bias, when backtesting, we must ensure our universe includes companies that were in the index historically even if they later got delisted or went bankrupt. If using something like the S&P 500, a survivorship-free dataset or index membership history is needed. We will incorporate that data (e.g., an index membership timeline) to know at each date which stocks are considered tradable. The ingestion will then fetch data for those (some might drop off later, which we handle). This is especially important if an LLM agent using news could inadvertently pick up that a company went bankrupt – we need the simulation to reflect that (the strategy hopefully exits before zero, but if not, it realizes the loss). Having a robust universe definition and data handling for defunct companies is part of our plan to make backtests realistic.

Adapting to New Markets Example: Suppose the team wants to try the strategy on **Forex** or **Futures**. The architecture supports it: ingestion would connect to an FX data source, technical features apply similarly, fundamental data might be macro data (for currencies, interest rates and economic indicators). LLM sentiment might parse central bank communications or news headlines about economies rather than company news. So we'd tweak the agents (maybe fine-tune an LLM on central bank statements for sentiment). The RL agent and HRM largely remain the same structure, just operating on different inputs. This underscores the advantage of modularity – each piece can be retooled for a different domain, while keeping the overall pipeline consistent.

In conclusion, we emphasize **U.S. equities** initially due to rich data and relevance. The design, however, is market-agnostic at its core, aligning with the idea of a platform that supports multiple markets (as

FinWorld's multi-market support demonstrates [43]). Stock universe selection is treated as an input to the system, not a hardcoded constant, allowing researchers to easily pivot and test across different universes to see where the strategy performs best.

Trading Frequency Considerations (Low-Frequency vs Intraday)

The suite is designed to handle multiple trading horizons, but different frequencies come with different pros, cons, and feasibility given current ML and LLM capabilities. Here we analyze how the pipeline can be configured for **low-frequency (weekly/monthly)** trading, **medium-frequency (daily)** trading, and **high-frequency (intraday)** trading, and what adjustments or expectations come with each:

1. Low-Frequency (Weekly/Monthly) Trading:
Low-frequency strategies involve holding positions for weeks, months, or even years – essentially longer-term investment strategies. Examples include factor investing (holding value stocks for months) or global macro trades that play out over quarters. For our pipeline: - *Pros:* A low-frequency approach aligns well with fundamental analysis and slower signals. LLM agents can deeply analyze quarterly reports, news trends over months, etc., without being rushed. The latency of LLMs is a non-issue here; we could spend minutes or hours generating a high-quality analysis for a monthly rebalance. Deep RL can be applied in principle (though with very few decision points, RL training might be data-starved – possibly one would use analytic methods instead). Also, transaction costs are low (few trades), so we can afford to be more thorough in decision-making.
- *Cons:* The number of training examples (trades) is extremely limited – if you rebalance monthly, 10 years is only ~120 data points for learning, which is generally insufficient for complex ML. Thus, purely data-driven methods (like training an RL policy from scratch) might not be reliable. Instead, we'd lean on human-designed rules supplemented by AI analysis. Another challenge is that long-term outcomes are influenced by many external factors that are hard to encode (economy, regime shifts). Our LLM's knowledge might actually shine here by connecting macro narrative to long-term prospects (e.g., LLM reading about a new technology trend and suggesting a long-term thematic play). But verifying an AI's suggestions over such long horizons is hard without forward testing for equally long periods.
- *Pipeline Adjustments:* We may not use the RL agent heavily for monthly trades; instead the HRM/aggregator might directly determine the allocation by reasoning (or we use simpler optimization like mean-variance on expected returns that the HRM/LLM estimates). The evaluation for low-frequency focuses more on fundamental performance (did we beat the index over years?) and we'd tolerate long periods of underperformance as long as thesis holds. Low frequency also means less data to update – ingestion can be end-of-week, and LLM analysis could be done leisurely. We might schedule the pipeline to run once a week with the latest data, produce a new portfolio, and that's it.

2. Medium-Frequency (Daily) Trading:
Daily trading (making decisions once per day, typically at market close or open) is a sweet spot for many quant strategies because it offers a decent number of sample points over years while still capturing short-term anomalies. - *Pros:* We have ~250 trading days a year, so over a decade ~2500 data points, which is somewhat reasonable for training ML models (especially if we also have cross-sectional data with many stocks each day – RL can learn from multiple stocks' experiences). Daily frequency is slow enough that we can use fairly heavy computations each day (like run an LLM on all news each morning) but fast enough to exploit news quickly (if we trade at next day open based on yesterday's news, that's still timely for many news items that have overnight effect). Many traditional factors (momentum, mean reversion, etc.) manifest at the daily level, so our engineered features and LLM insights (which update daily) can feed an RL or HRM

decision effectively. Execution is easier than intraday – we typically trade at the open or close price using market orders, which is straightforward and has high liquidity for large caps.
- *Cons:* There is still overnight risk (we hold positions that can gap). Some intraday info might be missed if we only check once daily (e.g., a news item at 10am – though our pipeline could be run more than once a day if needed). But primarily, daily frequency seems quite feasible with our tech. The latency of an LLM (couple seconds per stock perhaps) times a few hundred stocks is manageable if run concurrently or over the trading day. We just need results by market close (if trading at close) or by next open. There's a potential con in that many quant funds operate on daily signals; competition is higher in that space, so pure daily strategies may have lower Sharpe unless we incorporate unique data (like our LLM-driven sentiment). But that's an edge we aim for.
- *Pipeline Adjustments:* Running the pipeline daily means scheduling is important: e.g., fetch all new data after market close, run analysis agents early next morning, generate trades before open. We can automate this. The RL agent could be retrained periodically or use incremental learning as new data comes (maybe retrain each quarter on last few years of daily data to refresh). The evaluation can roll as well (we'll track a rolling window performance). We also might incorporate **overnight analysis** – e.g., have LLM do heavy analysis at night (when we have time) so that by morning we have results to trade on. Also, daily trading will highlight the importance of robust *"out-of-sample"* evaluation – we would use past years as train and every new day as essentially a test (a kind of walk-forward). We'll simulate that in backtest to ensure generalization.

3. High-Frequency (Intraday) Trading:
Intraday trading can range from a few trades per day to many trades per hour (we aren't considering ultra HFT like market-making on sub-second, which is beyond scope for LLM integration). - *Pros:* More data points for learning (intraday bars give thousands of points quickly), and the ability to capitalize on very short-lived inefficiencies (like mean reversion over minutes, or immediate reaction to news). If our pipeline can process news in real-time, we could trade off the immediate sentiment change. RL is naturally suited to sequential decision in intraday environments where it can get rapid feedback (winning or losing on each trade within hours). For example, an RL agent could learn a strategy like buy the dip and sell after rebound on minute bars, etc. Our signals like sentiment might give a bias ("market is bullish today, so bias long on dips"). Combining those could be potent. - *Cons:* As discussed, **latency** is a big issue. Intraday requires quick decision loops. If we trade on 5-minute bars, we have at most 300 seconds to gather data, compute features, and decide – in practice, we want to decide faster to execute within that bar. LLMs, especially large ones, are too slow to run on each bar. Also, the volume of text data intraday can be huge (hundreds of news articles or thousands of tweets per day for one stock). We likely can't run a full GPT-4 analysis on every single new tweet. Instead, we'd have to either drastically simplify (like use a smaller sentiment classifier model for intraday updates, possibly distilled from the LLM) or limit usage (like only react to **major** news via LLM, not every blip). Additionally, intraday has higher noise – ML models might overfit patterns that are just random. The RL agent could end up chasing noise and incur trading costs. Speaking of costs, intraday strategies have to contend with spread and commission on every trade – we must model these carefully. Execution becomes harder: moving large positions intraday can itself impact price, so our simulation must account for slippage and maybe RL has to learn to split orders. We also must maintain high reliability and uptime in a live intraday system – any downtime can miss trades. - *Pipeline Adjustments:* For intraday, we would streamline the pipeline: rather than re-running everything from scratch every minute (impossible with heavy models), we maintain a continuously updating system. Data ingestion is streaming, feature engineering is running in real-time (e.g., updating technical indicators tick by tick). LLM sentiment agent might run in a **parallel asynchronous loop**: e.g., whenever a new news item comes, quickly classify it using an already fine-tuned smaller model (like a MiniLM or even a keyword-based model initially for speed). We

might use the large LLM offline to improve the small model (knowledge distillation), but the live intraday sentiment feed would come from the small model for speed. The aggregator/HRM in intraday might not run every minute; it could run at a slower cadence or only when significant changes occur. Alternatively, HRM could be replaced by simpler logic intraday to avoid extra compute – e.g., an RL agent might directly incorporate multiple signals in its state and we rely on it. HRM could still run every hour to adjust high-level strategy and the RL fills details in between. This hybrid approach ensures we don't overload with reasoning at every step. Also, intraday RL likely would be trained on historical intraday data. One could use **Decision Transformers** here if we have a lot of sequential data, as they might capture patterns over multi-hour trajectories. But we must watch out for computational burden. Execution wise, we might deploy a different mode – e.g., connect to broker's websocket for live ticks and use a fast execution handler (maybe written in a faster language if needed, though Python might suffice if not extremely high freq).
- An example intraday scenario: At market open, our LLM aggregator sets a bias "market likely up today due to news ABC". Then, the RL agent throughout the day scalps or trades intraday using that bias, closing out by end of day. This way the heavy lifting by LLM is done once, not on every trade. Intraday decisions are then just numeric RL decisions which are very fast. This seems a feasible compromise.

To summarize frequency considerations: - **Low-frequency (weeks+):** Best for fundamentally-driven, LLM-heavy analysis, but limited ML training samples. We'll rely on HRM/LLM to justify and maybe simpler rules for decisions. - **Daily:** Balanced approach – enough data for ML, slow enough for LLM. Our primary focus likely, given current tech. Strategy can blend fundamental, sentiment, technical well here. - **Intraday:** Possible with careful architecture – require lighter models or pre-computed outputs, and leaning on RL for quick decisions. LLM integration must be judicious (maybe pre-market analysis or reacting only to large news). Emphasize low-latency engineering if implementing (multi-threading, etc.) and strong risk controls due to noise.

We should also note that strategies might differ by frequency: e.g., momentum and sentiment might be great on daily, whereas pure order book signals matter at HFT. We won't attempt to do ultra-HFT (like market making on the order book) because that requires microsecond latency and is far beyond LLM usage. Our intraday scope is more like "swing day trading" on minute/hour bars, which is still within a realm where Python and occasional model calls can work if optimized.

The trading suite's modular design allows choosing a frequency and adjusting modules accordingly. We could even run multiple instances of the pipeline concurrently at different frequencies (one sub-system does daily trades on portfolio, another does intraday tactical trades). They would then need some coordination (to avoid conflict, or one could trade a subset of capital intraday while core stays long-term). This is an advanced scenario, but interesting for future expansion – essentially a multi-frequency strategy where, say, long-term investment positions are decided weekly by fundamental HRM, and a portion of capital does intraday trades managed by RL.

In conclusion, **daily frequency** is likely the initial target as it offers the best trade-off given LLM and RL current capabilities. We'll ensure the architecture can later scale down (to weekly) or up (to intra-hour) by swapping components (like using faster models, altering training procedure, etc.). Trading frequency choice will also be guided by where our strategy shows edge: if the LLM sentiment edge decays after a day, daily trading is needed; if it persists weeks, we could trade slower. Our evaluation will include trying different rebalancing frequencies and observing performance.

Citations: The viability issues of using large models at high frequency are evidenced by cases like BloombergGPT's limitation in high-frequency trading due to cost/latency [6]. By contrast, smaller open models or no-LLM approaches are preferred intraday [10]. We incorporate these insights by matching strategy speed with appropriate tech (fast models for intraday, full LLM for daily+).

Uncertainty Estimation Module (True Confidence Estimation)

In any trading strategy, especially one driven by AI predictions, it's crucial to know **how confident** the model is in a given decision. To address this, our design includes an uncertainty estimation module, essentially a specialized classifier that outputs not just a buy/hold/sell decision but also a calibrated confidence level in that decision. The team's *"True Confidence Estimation"* method likely refers to a proprietary technique to accurately quantify prediction confidence – possibly using ensemble models, Bayesian approaches, or out-of-sample validation to gauge how much trust to place in a signal.

Module Description: The uncertainty estimation module will take as input the current market state and the signals the other agents have produced (and potentially the proposed action from the main strategy). It will output a refined classification like: - **Buy with High Confidence**, - **Buy with Low Confidence**, - **Hold/No-trade** (if things are uncertain or neutral), - **Sell with Low Confidence**, - **Sell with High Confidence**.

In simpler terms, it might output a probability or confidence score along with the action. For example: "80% confidence that 'Buy' is the correct action" versus "55% confidence". The *True Confidence* method presumably is a custom algorithm by the team, but we can integrate its output in a few ways: - If it's a standalone classifier (maybe an ML model trained to predict when our signals are right vs wrong), we can run it in parallel with the decision. For instance, it could be an ensemble of past predictions to actual outcomes that learned to detect conditions where predictions succeed or fail. Perhaps it looks at volatility, signal agreement, etc., and gives low confidence during volatile or conflicting signal times.
- If it's an analytical measure (like margin of safety), it could be computed from the distribution of model outputs. E.g., if our RL policy has Q-values for actions, an uncertainty measure could come from the spread between Q for buy vs sell (small spread means close call -> low confidence). Or an LLM might produce a "sure" answer vs a hedged one (we could parse the language like if HRM says "very likely" vs "possibly").

Integration with Decision Process: Once we have this confidence metric, it can be used to *modulate* the final trading decisions: - **Position Sizing:** We can tie position size to confidence. For instance, if our RL agent normally would go 100% long on a strong signal, but the confidence module says only 50% confidence, we might scale the position to 50%. Essentially, treat confidence as a multiplier or weight. This helps risk management: in cases of uncertainty, take smaller bets; in cases of high conviction, allow larger bets. This concept is akin to having a secondary model that predicts when your edge is big vs small, which is common in professional trading (sometimes called meta-labeling or Bayesian sizing).
- **Trade Filtering:** We could establish a confidence threshold below which trades are not taken at all. For example, if "True Confidence" < 60%, we decide to hold rather than trade. This prevents the system from churning on marginal signals. Especially in ML, often some trades have very slim expected edge – filtering them out can improve overall performance (trading only when probability of success is sufficiently above 50%). Our classifier could be tuned to identify those marginal cases and label them as "hold".
- **Switching Strategies:** The confidence module could even act at a higher level – if confidence is low across the board (like the model is unsure), perhaps we revert to a safe baseline strategy (like stay in cash or replicate the index) until confidence picks up. This ensures the system doesn't force trades in confusing regimes.

- **Risk Management:** A low confidence reading might trigger risk-off actions – e.g., reduce leverage, tighten stop losses. Alternatively, high confidence might allow a bit more risk tolerance for that trade. Essentially, it's an overlay that adjusts risk in real-time based on model's perceived reliability. This concept connects to Bayesian reasoning: if we're more certain in an outcome, we risk more on it (Kelly criterion style), if uncertain, risk less.

True Confidence Estimation Method: While the details of the team's method aren't specified, we can make educated guesses and incorporate it: - Possibly they employ an **ensemble of models** and measure agreement (if all models agree on buy, confidence high; if half say buy, half say sell, confidence low). In implementation, we could run multiple variant models in parallel – e.g., an LLM-based prediction, an RL model, a simpler regression – and see if they concur. Or ensemble just means retrain the RL or classifier multiple times and check variance.
- They might use **meta-modeling (Meta-Labeling)**: train a second-stage model on past trades, labeled 1 if trade was profitable and 0 if not, using features like initial signals, volatility, etc., to predict probability of success [44]. That probability becomes the confidence. For example, research in meta-labeling suggests you can often improve strategy by training a classifier to filter trades (this sounds exactly like what this module could be). If we have enough past trade data, we can do this: treat all our prospective trades in backtest as data, and train a model to predict which ones worked. Features for that could be: strength of sentiment signal, number of signals agreeing, time of day, previous performance streak, VIX level, etc. The output is probability that trade will be profitable above some threshold.
- They could incorporate **uncertainty quantification techniques** for deep models: e.g., Monte Carlo dropout or Bayesian neural nets to get a distribution of outputs. For instance, run our RL policy network with dropout 100 times and see distribution of action preferences – if it's tight (most runs say "buy"), confidence high; if 50/50 buy vs sell, confidence low. This is computationally heavier but maybe doable daily. Another approach: use the HRM or LLM to critique the trade – e.g., ask the HRM "how confident are you and why?" (though LLM self-reported confidence might not be reliable unless calibrated on data).

Output Utilization Examples:
- Let's say our strategy signals a Buy on AAPL today. The confidence module analyzes conditions and says confidence only 0.4 (40%). Perhaps the news is unclear or signals are conflicting. We then either decide not to buy (hold instead) or we buy but at a smaller size. If next day AAPL indeed doesn't perform, we avoided a loss or made it tiny – a win for the confidence filter.
- Conversely, if confidence was 0.9 on a trade, we might double the position compared to normal. If our confidence module is well-calibrated, trades with 0.9 confidence should win the vast majority of time. This can boost returns by allocating more to those. However, careful: we must ensure calibration, so that 90% confidence truly corresponds to ~90% success in long run, otherwise we could overshoot risk.

Integration with RL: We have two main ways to integrate: outside RL (as described above, as a filter or size adjuster after RL picks action) or inside RL. Inside RL, we could include the confidence as part of the state or reward. For example, feed the RL agent its own confidence – it might learn to only act strongly when confidence is high (though that might be redundant, easier just to filter externally). Or incorporate it into reward: maybe penalize actions taken when confidence was low that ended up losing (thus RL learns to avoid low-confidence moves). But a simpler pipeline is likely to do it as an overlay after the RL outputs initial action. That keeps the modules decoupled (RL focuses on maximizing reward assuming it must always act, and the confidence module then tones it down when needed).

Implementation Plan: We'll likely implement the True Confidence module as a separate classifier model that we train on historical strategy outputs vs outcomes (meta-labeling). In live operation, it takes current inputs and strategy signals and outputs a confidence score or label. The pipeline can then use that to adjust positions. Initially, we may start with a simple heuristic version (like based on volatility or agreement of signals) until we gather enough data to train a more complex model. Over time, as the strategy generates trades, we feed that into improving the confidence estimator (like an active learning loop). One must be cautious – meta-models can introduce their own overfitting. We will make sure to validate it on a separate set to confirm that a confidence threshold of say 70% indeed yields better-than-random outcomes historically.

Uncertainty in LLM outputs: Another angle is the LLM agents themselves can produce a confidence (for example, FinGPT might output sentiment logits, which we interpret as confidence of sentiment). We could feed these uncertainties into our confidence module. E.g., if sentiment is only weakly positive (LLM not very certain), maybe require another signal to align to trade. So the confidence module might consider the **strength of each signal**, not just final trade. HRM aggregator can also pass along if it found conflicting evidence, which indicates lower confidence. Essentially, we make sure the confidence estimator has all relevant info to judge when not to trust the model.

In summary, the Uncertainty Estimation module is a **safety valve and performance enhancer**. It aims to only deploy capital when the edge is sufficiently clear, using the team's True Confidence method to estimate that edge. By integrating this, we reduce drawdowns from bad or uncertain trades and focus on high-conviction opportunities, which should improve risk-adjusted returns. It adds an additional layer of intelligence to the pipeline, effectively making the strategy *self-aware* of its limitations on a trade-by-trade basis.

Research and Live Deployment Roadmap

Implementing this trading suite will be done in phases, transitioning from a pure research environment to live deployment in a controlled manner. Below is the roadmap with considerations at each stage, ensuring we validate the strategy thoroughly and address issues like data leakage and model bias before real money is at stake:

Phase 0: Offline Research & Prototyping
Before formal backtesting, we'll do extensive prototyping of each component on subsets of data. For example, test the LLM sentiment agent on a known sentiment dataset to ensure it's working, or run the RL algorithm on a simplified market model. This is to iron out technical bugs and get a sense of parameter sensitivities. No trading here, just component validation.

Phase 1: Historical Backtesting (Research Environment)
In this phase, we integrate the components and run **backtests on historical data**. We use Backtrader or a similar simulator with historical daily data (and intraday data if needed) to simulate how the strategy would have performed in the past. Key actions and precautions here: - We conduct backtests over multiple periods (e.g., 2015-2020 as one period, 2010-2014 another) to see consistency. We will also do **walk-forward testing**: e.g., train models on 2010-2015, test on 2016-2017, then retrain including 2016-2017, test on 2018, etc., mimicking how we'd update models in real life. This ensures our strategy isn't just curve-fitted to one time span. - **Avoiding Data Leakage:** A critical concern: our use of LLMs introduces a risk that the models might have seen future data in their training. For example, GPT-4 might "know" that a certain stock crashed

in 2020 due to its training data. If in backtest we prompt GPT-4 with news from 2019, it might implicitly use its knowledge of 2020 outcomes (even if not explicitly, through subtle cues). This could lead to over-optimistic backtest results (because the model appears prescient) [40] [39]. To mitigate this, we will **time-restrict LLM knowledge**: e.g., use an LLM with a known training cutoff (say 2021) and not test on data before 2021 as DeepFund suggests [41]. Or use techniques like "hide future info in prompts" by only giving it documents up to that date (which we anyway do). We may also consider fine-tuning our own LLM on data up to certain year and ensure not beyond. The evaluation module will specifically look for signs of leakage (like strategy doing unusually well right before known historical turning points – a red flag that maybe the model knew). If identified, we adjust approach (maybe remove that period from test or retrain model without those data). - We verify that **preprocessing and feature generation are all done in a rolling, causal way** in backtest. For example, when computing a 1-month momentum as of Jan 2018, we ensure it only uses data up to Jan 2018, not any future data. We also simulate publication delays (e.g., if using an earnings report that came out on Feb 15, the strategy can only use it after Feb 15, not on Feb 1). These are fine details but crucial for realistic backtests. - We will run **many backtest scenarios**: vary parameters, try excluding each agent to see its impact (ablation tests), test different market regimes (bull vs bear periods separately) to see how the strategy behaves. This gives insight into strengths and weaknesses. - If the backtest results look too good to be true, we scrutinize for overfitting or leakage. For instance, if we see perfectly smooth equity curve, likely something is fishy. We might do randomization tests – e.g., shuffle certain inputs to confirm performance drops (if it doesn't, something's wrong because the signals became nonsense but strategy still did well).
- During backtesting, we also measure things like turnover, average holding period, and strategy capacity (how much capital it could deploy without impacting market significantly). These practical stats help in thinking ahead to deployment (e.g., if strategy trades 100% of volume of some stock, that's not feasible live). - We use the **evaluation metrics** described to identify if performance is robust (e.g., statistically significant alpha, reasonable Sharpe). If results are only marginal (like Sharpe ~1 with lots of volatility), we either refine the approach or accept that this might need improvement before live.

Backtesting is iterative. We likely will tweak the strategy multiple times here. However, we must be careful not to over-optimize on the backtest data (a classic pitfall). Every tweak based on backtest could be weaving in bias. To mitigate that, we can reserve a final test period that we do not touch until the very end. For example, use data until 2022 to develop, and keep 2023 aside entirely. Only once we lock the strategy, we run it on 2023 data as a pseudo "live forward-test" to see if it generalizes. This mimics doing a paper trade in historical hindsight.

Phase 2: Paper Trading (Forward Testing with Live Data)
After we gain confidence in backtests, we move to **paper trading** – running the strategy live on current market data but in a simulated account (no real money). This phase is crucial to see how our system operates in real-time and to catch any issues that don't show up in backtest: - We will deploy the pipeline (possibly on a cloud server or robust machine) to run continuously or daily as needed, connected to a broker's paper trading API (like Alpaca Paper). The strategy will receive live market feeds (or frequent polling) and make decisions which result in simulated trades.
- We closely monitor performance here and compare it to backtest expectations. If our backtest indicated, say, a 0.1% daily profit expectation, but in paper trading after 1 month we are at -2%, we investigate. Paper trading can reveal **implementation issues**: e.g., maybe our live data feed has slight differences or delays causing suboptimal entries, or our model decisions come a bit too late. It can also reveal **market changes**: perhaps regime shifted and our strategy needs adaptation.
- We also pay attention to **transaction costs and slippage** in paper trading. Often, backtests assume a

certain fill price (e.g., next open). In live, maybe we get a bit worse price. Alpaca's paper trading tries to simulate fills, but it might fill everything at last price which could be unrealistic for large orders. We can adjust our execution logic if we see partial fills or delays. Paper trading results help calibrate our slippage models – we can feed that data back into backtest to improve realism (closing the loop between research and reality).
- Paper trading also tests the **infrastructure**: can our pipeline fetch data and make decisions within the required time? Are there any crashes or memory issues when running continuously? Does the LLM agent blow up with some unexpected input? We will harden the system in this phase by adding error handling, fallback strategies (if an agent fails, maybe default to some safe action rather than no action), etc. We likely run the system for several weeks to a few months on paper to collect enough sample trades. - **Data leakage concerns in live:** In live forward testing, the leakage issue diminishes – the models don't have future data because the future hasn't happened. However, one subtlety is if we used a pretrained LLM (like GPT-4) that was trained up to 2021, it might not know about very recent entities (like a company that IPO'd in 2022 or a new term). That could be a problem in live: the LLM might be outdated. We may need to address that by supplying relevant context to LLM or fine-tuning it on more recent data before live. But fine-tuning on recent data risks that it "knows" recent history when we were backtesting – a tricky balance. Since we're now forward, updating models is fine as long as we don't use them to re-trade the past. So likely, before paper trading, we update any models to have knowledge up to the start of paper (e.g., fine-tune LLM on 2022-2023 news if needed) so they aren't ignorant of current facts. This is acceptable because we're no longer evaluating on that period. - We validate the **confidence module** in paper: see if our "True Confidence" predictions align with realized outcomes. If not, recalibrate. Because in live, we can see in real-time: "We had high confidence on this trade, but it failed," which might indicate mis-calibration. We may tweak thresholds or retrain the confidence model periodically using the new data. - The **HRM explanations** during paper trading will also be monitored. Are they making sense? If HRM says "buy because X and Y" and the market falls because of Z (which it didn't consider), maybe we need to feed more info to HRM. This is how we iterate to improve reasoning coverage.

At the end of a successful paper trading phase, we expect to see that performance metrics roughly match backtest (accounting for some decay due to real friction), and no major unexpected issues are left. It's also a time to get comfortable with operations – e.g., ensuring alerts are set up if something goes wrong (we don't want it placing wild orders unchecked, even in paper).

Phase 3: Gradual Live Deployment (with Real Capital)
Once the strategy proves itself in paper trading, we proceed to live trading with real money, but in a phased manner: - We'll likely start with a **small capital allocation** to test the waters. For instance, trade at 10% of intended size or with a very small account. The goal is to see if there are any differences in real execution vs paper. Sometimes paper trading doesn't account for all fees or liquidity impacts, so even a small live test can surface those (e.g., maybe you get partial fills or price impact that paper didn't simulate). - During early live trading, we intensively monitor performance and also system stability. If any anomaly is spotted (like strategy losing much more than expected or weird trades happening), we can quickly pull back. Ideally, we have built in some **kill-switches**: e.g., if cumulative P&L drops more than X% from high, system stops trading (could indicate something broke or regime shift). - **Risk management** is paramount now: we double-check all risk limits in code (max position, etc.) to ensure no runaway trades. We might also use broker safeguards (like max order size limits). - We continue running evaluation and logging as before. All trades are logged with their signals and rationales (for post-mortem). This helps in weekly or monthly reviews to see if the strategy is behaving as intended. - Over time, if small-scale live results are good (matching paper/backtest), we can scale up the capital to the desired level. We'll also continuously

incorporate new data into our models: e.g., retrain the RL agent on the latest data every so often (but carefully, not to overfit to recent noise – maybe use expanding window retraining). - We'll keep the **paper trading environment running in parallel** for testing any modifications. For example, if we want to upgrade the LLM model or try a new feature, we'd test that in paper mode while the original version trades live, until confident to switch. This parallel approach ensures any improvements are validated before affecting real money. - As live trading continues, we pay attention to **market regime changes**: if something like 2020 COVID crash happens (extreme outlier event), does our system handle it? We likely simulate some such scenarios in backtest, but reality can be different. The HRM reasoning might help here by recognizing unprecedented conditions and perhaps advising caution (one could even program HRM with a rule like "if regime is totally unlike training data, reduce confidence"). But overall, human oversight remains important in early deployment.

Limitations of Backtesting with Pretrained LLMs: It's worth re-emphasizing that one fundamental limit of backtesting our approach is the pretrained knowledge issue. We have addressed it by careful evaluation splits [39] and forward testing, but in general, an LLM-based strategy might always have a bit of *"informational advantage"* in historical tests that wouldn't exist going forward. For example, an LLM trained on 2021 data might implicitly carry information about relationships that became obvious by 2021 but might not have been obvious in 2015. When we backtest on 2015-2020, the LLM might exploit that (like knowing a company's long-term outcome). To combat that, our forward-testing/paper phase is the ultimate judge – that's truly how it behaves when it doesn't know the future. This is why we don't skip the paper phase. And we'll emphasize to stakeholders that backtest results, especially with AI, should be taken with caution and we rely on forward performance to validate [40] .

Continuous Learning and Deployment: After initial deployment, the roadmap continues with maintenance: monitoring, updating models periodically (perhaps using an automated retraining pipeline if appropriate, but with safeguards), and researching improvements as market evolves. We also plan for a feedback loop: e.g., incorporate live trade outcomes to refine the confidence estimator (true confidence module gets smarter with more data), possibly do **reinforcement learning fine-tuning in live** (like an online RL scheme, though one must be careful to avoid drift). FinWorld's concept of "RL-based finetuning for LLMs and agents" hints that continuously improving the models with new data is beneficial [23] , which we adopt, albeit making sure to always test changes in a research environment first.

In summary, our deployment approach is **incremental and cautious**: 1. Rigorous backtest (with anti-leak measures), 2. Then contained forward test in simulation (paper), 3. Then stepwise live introduction with risk limits, 4. Ongoing monitoring and improvement.

By following this roadmap, we aim to ensure the trading suite is robust, not overfit, and adapts well from the lab to the real world. The emphasis on forward-testing echoes best practices in algorithmic trading – as the saying goes, *"Past performance does not guarantee future results,"* so only forward testing can truly prove the strategy's merit [45] . We use backtests to develop and filter ideas, but we rely on careful forward evaluation to justify live deployment.

Throughout this document, we integrated insights from academic and industry sources to guide our design decisions, ensuring the system leverages cutting-edge techniques (like HRM reasoning [27] , LLM-based factor mining [1] , and deep RL [24]) while acknowledging practical constraints (LLM latency [6] , data leakage pitfalls [39] , need for interpretability [35] , etc.). By combining these components in a modular architecture, the suite stands as a modern research-oriented trading platform that can be iteratively improved and safely ushered from research to real trading. The mermaid diagrams provided illustrate how these pieces fit together, and the structured markdown format (with collapsible sections and clear headings) is intended to make it easy for our team to navigate this design in an Obsidian workspace as we progress from concept to implementation to deployment.

---

[1] [2] [17] [18] [20] [36] [37] AlphaAgent: LLM-Driven Alpha Mining with Regularized Exploration to Counteract Alpha Decay
https://arxiv.org/html/2502.16789v2

[3] [4] [5] [9] [10] Comparison of Open-Source and Proprietary LLMs for Machine Reading Comprehension: A Practical Analysis for Industrial Applications
https://arxiv.org/html/2406.13713v2

[6] [7] [11] [12] [13] [14] [15] [16] [33] [38] An End-To-End LLM Enhanced Trading System
https://arxiv.org/html/2502.01574v1

[8] [2503.16252] Fin-R1: A Large Language Model for Financial Reasoning through Reinforcement Learning
https://arxiv.org/abs/2503.16252

[19] Benchmarking deep reinforcement learning approaches to trade ...
https://www.sciencedirect.com/science/article/abs/pii/S0927538X25002136

[21] [24] [25] An Overview of Machine Learning, Deep Learning, and Reinforcement Learning-Based Techniques in Quantitative Finance: Recent Progress and Challenges
https://www.mdpi.com/2076-3417/13/3/1956

[22] [23] [43] FinWorld: An All-in-One Open-Source Platform for End-to-End Financial AI Research and Deployment - haebom
https://slashpage.com/haebom/daily_arxiv?lang=en&tl=en&post=dwy5rvmjg89yn2p46zn9

[26] [27] [35] The Hierarchical Reasoning Model (HRM): The Brain-Inspired AI Breakthrough You Need to Know About in 2025 | by Souvik Misra | Aug, 2025 | Medium
https://medium.com/@svkmsr6/the-hierarchical-reasoning-model-hrm-the-brain-inspired-ai-breakthrough-you-need-to-know-about-28e37a3b3c97

[28] [29] [30] [31] Daily Papers - Hugging Face
https://huggingface.co/papers?q=hierarchical%20reasoning%20architecture

[32] Beyond the Black Box: A Practical Playbook for Explainable LLMs
https://arthirajendran.medium.com/beyond-the-black-box-a-practical-playbook-for-explainable-llms-405f64ce2a98

[34] Designing Heterogeneous LLM Agents for Financial Sentiment ...
https://www.researchgate.net/publication/383103703_Designing_Heterogeneous_LLM_Agents_for_Financial_Sentiment_Analysis

[39] [40] [41] [45] DeepFund: Will LLM be Professional at Fund Investment? A Live Arena Perspective
https://arxiv.org/html/2503.18313v1

[42] quant-finance-lectures - Universe Selection - Code Library
https://www.quantrocket.com/codeload/quant-finance-lectures/quant_finance_lectures/Lecture29-Universe-Selection.ipynb.html

[44] When AI Knows Too Much: Look-Ahead Bias in Decisions
https://ashokreddy.substack.com/p/when-ai-knows-too-much-look-ahead