Cognitive Alpha Mining via LLM-Driven Code-Based Evolution

Anonymous ACL submission

Abstract

Discovering effective predictive signals, or "alphas," from financial data with high dimensionality and extremely low signal-to-noise ratio remains a difficult open problem. Despite progress in deep learning, genetic programming, and, more recently, large language model (LLM)–based factor generation, existing approaches still explore only a narrow region of the vast alpha search space. Neural models tend to produce opaque and fragile patterns, while symbolic or formula-based methods often yield redundant or economically ungrounded expressions that generalize poorly. Although different in form, these paradigms share a key limitation: none can conduct broad, structured, and human-like exploration that balances logical consistency with creative leaps. To address this gap, we introduce the *Cognitive Alpha Mining Framework (CogAlpha)*, which combines code-level alpha representation with LLM-driven reasoning and evolutionary search. Treating LLMs as adaptive cognitive agents, our framework iteratively refines, mutates, and recombines alpha candidates through multi-stage prompts and financial feedback. This synergistic design enables deeper thinking, richer structural diversity, and economically interpretable alpha discovery, while greatly expanding the effective search space. Experiments on 5 stock datasets from 3 stock markets demonstrate that CogAlpha consistently discovers alphas with superior predictive accuracy, robustness, and generalization over existing methods. Our results highlight the promise of aligning evolutionary optimization with LLM-based reasoning for automated and explainable alpha discovery.

1 Introduction

Alpha mining is the process of discovering predictive financial signals, or "alphas," from financial markets such as the stock market to forecast future asset returns. However, since financial markets are characterized by high dimensionality, time-varying volatility (Engle, 1982), and a low signal-to-noise ratio, it remains challenging to identify explainable, reliable, and diverse alphas that support sustainable profitability and effective risk management. Over the decades, alpha mining has undergone several major transformations: from manual construction, to machine learning–driven automation, and more recently, to generative and reasoning-based exploration using LLMs (Guo et al., 2024a).

In the earliest stage, alpha factors were manually designed by financial experts, grounded in economic intuition and empirical observation. Classic examples include the Fama–French factors (Fama and French, 1992) and various documented financial anomalies (Harvey et al., 2016; Hou et al., 2017). These human-crafted alphas are interpretable and theoretically sound. However, the design process is inherently labor-intensive and inefficient. As financial markets became increasingly complex and data-rich, manual approaches struggled to scale, resulting in diminishing returns and crowding among similar strategies.

To enhance efficiency, researchers began leveraging machine learning models for alpha discovery. Some studies directly employed neural networks (Duan et al., 2022; Xu et al., 2021a,b) to implicitly extract complex and nonlinear alpha structures from market data through deep learning. These neural approaches demonstrate strong predictive power and the ability to capture high-dimensional and nonlinear dependencies. However, they also suffer from inherent weaknesses: such models often behave as black boxes, making it difficult to trace the underlying decision logic or assess their robustness under changing market conditions. As a result, their performance tends to degrade when exposed to regime shifts or unseen patterns. In contrast, formula-based approaches (Zhang et al., 2020, 2023) aim to identify alphas represented by explicit mathematical expressions. Many methods based on genetic programming (GP) (Cui et al.,

2021; Lin et al., 2019; Patil, 2023; Schmidt and Lipson, 2010; Su et al., 2022) and reinforcement learning (RL) (Liu et al., 2021; Yu et al., 2023; Shi et al., 2025a) frameworks have been proposed to automatically search symbolic formula spaces. These methods provide transparent expressions that are easy to reproduce and evaluate. Nonetheless, the resulting formulas are often overly complex or redundant and frequently lack solid economic or financial rationale, causing weak generalization and limited stability in real trading environments. Despite their differences, both neural and formula-based paradigms share a common limitation: their search processes are inefficient and narrow in scope. Neither can emulate human-like reasoning that combines logical consistency with leap-style creativity, leaving a critical gap between algorithmic exploration and genuine conceptual innovation.

Recently, LLMs (Li et al., 2025) have been introduced into alpha mining due to their knowledge integration, abstraction, and generative reasoning capabilities. LLMs can synthesize financial knowledge and propose novel formulaic representations at scale. Nevertheless, most existing LLM-based approaches (Shi et al., 2025b; Tang et al., 2025) still rely on formula stacking and pattern repetition rather than genuine reasoning or structural innovation. As a result, the generated factors tend to be redundant and susceptible to crowding effects, which limits their sustainability in dynamic market environments. The key research gap lies in how to evolve LLMs from mere *pattern replicators* into genuine *cognitive thinkers*. Specifically, there remains an unmet need for frameworks that enable LLMs to perform deeper thinking, richer structural diversity, and economically grounded exploration, thereby improving the long-term stability and robustness of the discovered alpha factors. Achieving this would move the field beyond brute-force search or shallow formula generation toward a more knowledge-driven and explainable paradigm for alpha discovery.

To bridge this gap, we propose a novel framework named **CogAlpha** (*Cognitive Alpha Mining*). The name highlights two key aspects of our approach: *Cognitive* and *Alpha*. The term *Cognitive* reflects the deeper reasoning capability of large language models, which allows the framework to go beyond shallow pattern recognition and engage in human-like analytical thinking. The term *Alpha* corresponds to the central goal of discovering profitable signals in quantitative finance. By integrating an evolutionary search process that induces deeper thinking in LLMs, together with a seven-level agent hierarchy and a multi-agent quality checker, CogAlpha naturally embodies our vision of advancing toward Cognitive Alpha Mining.

The remainder of this paper is organized as follows. Section 2 reviews related work on LLM-driven alpha mining and deeper LLM thinking. Section 3 presents the proposed CogAlpha framework in detail, highlighting its seven-level agent hierarchy, multi-agent quality checker, and thinking evolution components. Section 4 states the experimental setting and reports experimental results on five different stock datasets, with a primary focus on the CSI300, demonstrating the superiority of our approach. Section 5 concludes the paper and outlines promising directions for future research.

The main contributions of this work are summarized as follows:

- We introduce the concept of *Cognitive Alpha Mining*, which opens a new direction for automated, robust, and explainable alpha discovery, and we formalize it through the proposed CogAlpha framework.

- We propose a novel method, CogAlpha, which leverages an evolutionary search process that induces deeper thinking in LLMs, together with *a Seven-Level Agent Hierarchy* and *a Multi-Agent Quality Checker*.

- Extensive experiments on five datasets from 3 stock markets demonstrate the effectiveness of CogAlpha framework. The alphas extracted by our method exhibit stronger predictive performance, greater stability, and improved interpretability compared with baselines.

2 Related Work

Alpha Mining with LLM Alpha mining is a fundamental task in quantitative finance, aimed at discovering predictive signals, i.e., alpha factors, for stock markets. Previous approaches have primarily relied on human experts (Fama and French, 1992), genetic programming (GP) (Cui et al., 2021; Lin et al., 2019; Patil, 2023; Schmidt and Lipson, 2010; Su et al., 2022), reinforcement learning (RL) (Liu et al., 2021; Yu et al., 2023; Shi et al., 2025a), or deep learning (Duan et al., 2022; Xu et al., 2021a,b) to explore the vast factor space. However, these methods all have inherent limitations: they may be

inefficient, produce overly complex solutions, or suffer from limited interpretability.

Recently, LLMs, with their extensive world knowledge and strong reasoning capabilities, have been introduced into alpha mining. For example, AutoAlpha (Kou et al., 2024) employs LLMs to evaluate and select superior alpha candidates, while agentic frameworks have been incorporated to enhance adaptivity and automation. AlphaAgent (Tang et al., 2025) introduces an agent-based architecture with regularization strategies to mine decay-resistant alpha factors, AlphaJungle (Shi et al., 2025b) presents an LLM-powered Monte Carlo Tree Search (MCTS) framework in which the LLM performs multi-step formula refinement, and RD-Agent(Q) (Li et al., 2025) proposes a data-centric feedback loop with factor–model co-optimization that enables continuous factor adaptation under dynamic market conditions. Nevertheless, most existing LLM-based alpha mining methods remain constrained to formulaic searches, exploring only shallow regions of the factor space and failing to fully utilize the coding and reasoning capabilities of LLMs. Unlike prior work, we leverage the knowledge, coding proficiency, and reasoning ability of LLMs to explore alphas within a broader and deeper search space.

Evolving LLM Thinking To further explore the potential of large language models (LLMs), numerous methods have been proposed to enhance their thinking and reasoning capabilities. Recent studies have investigated integrating genetic and evolutionary algorithms (EAs) with LLMs. For example, Mind Evolution (Lee et al., 2025) employs an evolutionary search strategy to scale inference-time computation in large language models. WizardLM (Xu et al., 2024) enhances LLM performance by automatically generating large volumes of open-domain instructions across diverse topics and difficulty levels. EvoPrompt (Guo et al., 2024b) combines evolutionary algorithms with LLMs to optimize prompts using operators such as initialization, selection, crossover, mutation, and evaluation, all guided by an LLM; this approach outperforms both human-designed and traditional automated prompts. FunSearch (Romera-Paredes et al., 2024) applies an LLM-guided evolutionary search to discover mathematical heuristics, excelling in constructing novel mathematical objects and advancing algorithmic discovery. AlphaEvolve (Novikov et al., 2025) further scales this idea by introducing an autonomous evolutionary coding pipeline, where LLMs generate code variants and evaluators iteratively assess and refine them. Beyond these studies, evolutionary approaches combined with LLMs have also been explored in text generation (Xiao and Chen, 2023; Jobanputra et al., 2025) and code generation (Pinna et al., 2024; Hemberg et al., 2024). Despite these advances, none of the existing works specifically focus on extracting effective signals from highly volatile financial markets. To this end, we propose CogAlpha, which leverages an evolutionary search process that induces deeper thinking in LLMs, in collaboration with a seven-level agent hierarchy and a multi-agent quality checker, to generate robust and interpretable alpha factors.

3 Approach

The Cognitive Alpha Mining Framework (CogAlpha) is designed to simulate human-like reasoning and discover more sophisticated, logical, and interpretable alpha solutions. It employs an evolutionary search strategy that induces deeper thinking in LLMs, together with a seven-level agent hierarchy and a multi-agent quality checker, to perform alpha mining. Each alpha produced by **CogAlpha** is accompanied by detailed comments that explain its logic, clarify its underlying idea, and present the corresponding formula. Following the comments, the implementation code is provided. In this section, we introduce the core components of CogAlpha and explain how each part functions within the overall framework.

3.1 Seven-Level Agent Hierarchy

The only raw factors available are *open*, *high*, *low*, *close*, and *volume* (OHLCV). Based on the five factors, we design a seven-level agent hierarchy to explore alphas as comprehensively as possible. This hierarchy consists of 21 unique agents. As shown in Figure 2, from a macroscopic (Level I) to a microscopic (Level VII) perspective, these agents are organized into seven hierarchical levels. Each agent is dedicated to exploring a distinct alpha-discovery direction and independently generates a set of alpha factors according to its designated exploration strategy. The following provides a brief overview of each level's exploration domain, and more details are provided in Appendix A.1.

Level I: Market Structure & Cycle Layer (*AgentMarketCycle*, *AgentVolatilityRegime*) — Ex-

plores large-scale temporal structures such as long-term trends, market phases, and cyclical state transitions inferred from daily OHLCV dynamics.

Level II: Extreme Risk & Fragility Layer (*AgentTailRisk*, *AgentCrashPredictor*) — Models tail-risk exposure, crash precursors, and systemic fragility patterns that indicate potential regime breakdowns or stress accumulation.

Level III: Price–Volume Dynamics Layer (*AgentLiquidity*, *AgentOrderImbalance*, *AgentPriceVolumeCoherence*, *AgentVolumeStructure*) — Captures the interaction between price and trading activity—liquidity, order imbalance, and coherence between price movement and volume behavior.

Level IV: Price–Volatility Behavior Layer (*AgentDailyTrend*, *AgentReversal*, *AgentRangeVol*, *AgentLagResponse*, *AgentVolAsymmetry*) — Analyzes trend persistence, short-term reversal, volatility clustering, and asymmetric price dynamics as the core source of predictive alpha.

Level V: Multi-Scale Complexity Layer (*AgentDrawdown*, *AgentFractal*) — Measures cross-scale irregularity, fractal roughness, drawdown–recovery geometry, and long-memory characteristics in time-series structure.

Level VI: Stability & Regime-Gating Layer (*AgentRegimeGating*, *AgentStability*) — Assesses temporal stability and constructs adaptive gating mechanisms that regulate signal activation under varying market conditions.

Level VII: Geometric & Fusion Layer (*AgentBarShape*, *AgentCreative*, *AgentComposite*, *AgentHerding*) — Focuses on geometric pattern representation (candlestick morphology) and multi-factor fusion, combining independent signals into coherent composites.

3.2 Diversified Guidance

To achieve more precise and comprehensive exploration along each alpha-discovery direction, we extend the original guidance generation with five paraphrasing modes: *light*, *moderate*, *creative*, *divergent*, and *concrete*. The *light* version performs minimal rewording to maintain almost identical meaning, ensuring linguistic consistency for baseline comparison. The *moderate* version introduces natural phrasing variations to enrich expression while keeping the same analytical focus. The *creative* version adds interpretative depth and research-oriented nuance to inspire alternative reasoning within the same conceptual boundary. The *divergent* version explores new but related analytical perspectives, helping generate complementary hypotheses beyond the original phrasing. Finally, the *concrete* version transforms abstract descriptions into measurable, implementation-oriented forms by specifying possible formulas, ratios, or statistical operations. Together, these five paraphrasing styles enable broader semantic coverage and deeper factor reasoning without departing from the original analytical intent. More details about those paraphrasing modes are provided in Appendix A.2.

3.3 Multi-Agent Quality Checker

To verify the validity and quality of the generated alpha codes, we design a *Multi-Agent Quality Checker*—comprising the *Judge Agent*, *Logic Improvement Agent*, *Code Quality Agent*, and *Code Repair Agent*. All alpha codes that pass the quality checker are stored in the candidate pool; otherwise, invalid codes are sent back to the multi-agent system for repair. Codes that cannot be repaired or improved after several attempts are discarded.

As illustrated in Figure 1, the *Code Quality Agent* first detects issues such as syntax errors, formatting inconsistencies, and runtime bugs. If such issues are found, the *Code Repair Agent* attempts to fix the problematic alpha codes based on the feedback provided by the *Code Quality Agent*. Next, the *Judge Agent* evaluates whether an alpha factor is logically consistent, technically correct, and economically meaningful. If improvement is needed, the *Logic Improvement Agent* refines and enhances alpha codes that fail the *Judge Agent*'s assessment. After passing all quality checks, each code is executed. We evaluate numerical stability by detecting runtime errors, the proportion of *NaN* values, overflow/underflow, and distinct values per day. Codes that fail are either rejected or sent back to earlier agents for correction. If it runs successfully, a unit test is performed to examine potential information leakage. Codes that pass the unit test are deemed qualified and stored in the candidate pool. More details are shown in Appendix A.3.

3.4 Fitness Evaluation

After passing the Multi-Agent Quality Checker, each alpha is evaluated using five predictive power metrics: Information Coefficient (**IC**), Information Coefficient Information Ratio (**ICIR**), Rank Information Coefficient (**RankIC**), Rank Information

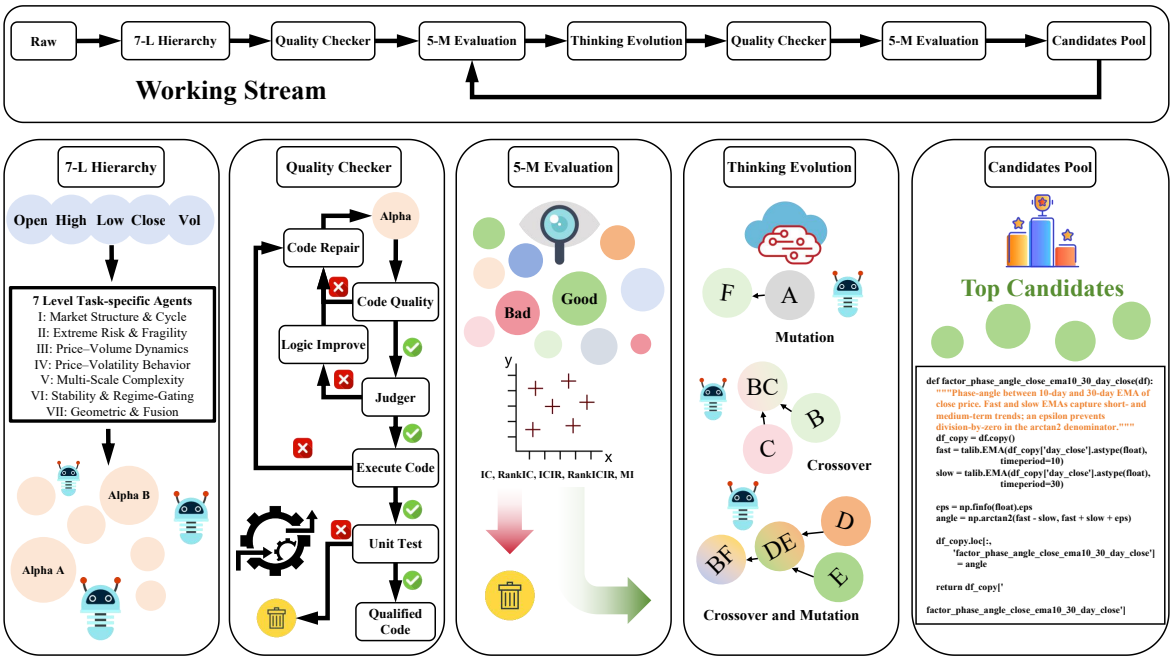

Figure 1: **Overview of CogAlpha.** The Seven-Level Agent Hierarchy produces initial alphas derived from the OHLCV data. The Multi-Agent Quality Checker verifies the validity and quality of each generated alpha code. The Filtering module evaluates all alpha codes using five predictive power metrics. Finally, the Thinking Evolution module iteratively refines and recombines qualified candidates through deeper reasoning by LLMs in each iteration.

Coefficient Information Ratio (**RankICIR**), and Mutual Information (**MI**). The first 4 metrics measure the linear relationship between the alpha and the target return, whereas **MI** captures the nonlinear dependency between them. Detailed definitions of these metrics are provided in Appendix B.3.

We set threshold values to identify *qualified* and *elite* alphas. Alphas whose five evaluation metrics all exceed the 65th percentile among all alphas in the same generation are classified as *qualified alphas*, while those exceeding the 80th percentile are considered *elite alphas*. We constrain each metric by a minimum bound to prevent the dominance of outliers: *IC* and *RankIC* are bounded below by 0.005, *ICIR* and *RankICIR* by 0.05, and *MI* by 0.02. For elite factors, the minimum bounds are slightly higher: 0.01 for *IC* and *RankIC*, 0.1 for *ICIR* and *RankICIR*, and 0.02 for *MI*. The qualified alphas form the new parent pool and are passed to the next iteration, whereas the elite alphas are stored in the final candidate pool. Additionally, the top two elite alphas from the previous generation are carried forward to the next generation to preserve high-quality solutions.

3.5 Adaptive Generation

After each fitness evaluation, there exists a population of valid alphas and another of invalid alphas, each with different underlying causes. To ensure that agents can continuously learn from previous generations, information about both valid and invalid alphas is incorporated into the prompt. For each generation, we randomly select two valid alphas and two worst-performing invalid alphas as guiding samples. Each selected alpha is first analyzed and summarized to explain why it is valid or invalid. Subsequently, the combined fitness results and analytical summaries of the selected alphas are incorporated into the generation prompts, based on which new alphas are generated.

3.6 Thinking Evolution

To guide the LLM to reason more deeply about alpha searching, we employ *Thinking Evolution* to enhance its alpha-mining capability. All qualified alphas are evolved through this process. As illustrated in Figure 1, *Thinking Evolution* implements a genetic-style optimization process in natural language space, where candidate alpha codes undergo mutation and crossover operations expressed through textual prompts. It consists of 2 agents: the *Mutation Agent* and the *Crossover Agent*. The *Mutation Agent* slightly modifies a given alpha code to introduce variability, whereas the *Crossover Agent* generates a new alpha code by combining two existing ones. Three types of evolution are conducted: mutation only, crossover only, and crossover followed by mutation. After each evolution step, the

resulting alpha codes are examined by the Multi-Agent Quality Checker. This process continues until all generations are completed.

4 Experiments

In this section, we first describe the experimental settings and compare our framework with the baselines. Then, we demonstrate the interpretability and evolutionary process of the generated alphas. Finally, we study the sensitivity of our method to different metric thresholds.

4.1 Experimental Settings

Datasets Our experiments are mainly conducted on the CSI300 (China Securities Index 300). Its components consist of 300 large-cap A-share stocks in the Chinese market. We primarily use the 10-day return as the prediction target, with buying and selling at the open price. The dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2024/12/01) periods. Four other datasets (CSI500, S&P500, HSI, and HSCI) from three different stock markets (China, U.S. and HK) are also tested. More details and results are provided in Appendix B.1 and Appendix B.8.

Model In our paper, all agents are based on **gpt-oss-120b** (OpenAI, 2025a) by default. For the task-specific agents in the seven-level agent hierarchy and the thinking-evolution agents, the temperature is randomly selected from {0.7, 0.8, 0.9, 1.0, 1.1, 1.2} to encourage diversity. For the agents in the multi-agent quality checker, the temperature is fixed at 0.8. The maximum token length is set to 4096. By default, LightGBM (Ke et al., 2017) is used to train the alphas generated by our method.

Training Setting The size of the initial pool is set to 80, meaning that the minimum number of alphas generated by the task-specific agents is 80. The parent pool size is set to 32, indicating that after filtering, at most 32 alphas are retained and passed to the next generation. The children's pool is set to be three times the size of the parent pool, meaning that the minimum number of alphas generated by the evolutionary agents is 96. By default, each task-specific agent leads a complete evolutionary cycle, which consists of 24 generations and 3 inner sub-cycles, with each sub-cycle comprising 8 generations. Thus, each task-specific agent initiates the evolutionary search 3 times. In addition, every 2 generations, new alphas generated by the task-specific agents are filtered and injected into the parent pool. For each generation, the top two elite alphas from the previous generation are always carried forward to the next. All alphas containing more than 30% NaN values or failing the multi-agent quality checker are discarded. All experiments are conducted on NVIDIA H100 GPUs.

Evaluation We use four predictive power metrics to evaluate the performance of alpha combinations: the Information Coefficient (**IC**), Information Coefficient Information Ratio (**ICIR**), Rank Information Coefficient (**RankIC**), and Rank Information Coefficient Information Ratio (**RankICIR**). The **IC** measures the linear correlation between alpha values and subsequent total returns, reflecting the overall predictive power of the alpha. The **ICIR** quantifies the stability and temporal consistency of **IC**. The **RankIC** and **RankICIR** are similar to **IC** and **ICIR**, respectively, but they measure the monotonic relationship between the alpha and subsequent total returns rather than linear correlation. In addition, two performance indicators are employed: the Information Ratio (**IR**) and Annualized Excess Return (**AER**). The **IR** evaluates the risk-adjusted excess return, and the **AER** measures the annualized excess cumulative return over a given period. Detailed definitions and formulas of these metrics are provided in Appendix B.3.

4.2 Comparison with Baselines

We conduct a comprehensive evaluation of CoGAlpha by comparing it against 21 benchmark methods from various application domains (see Table 1). First, we select 7 commonly used machine learning models in quantitative finance: Linear Regression, MLP, Random Forest (Breiman, 2001), LightGBM (Ke et al., 2017), XGBoost (Chen and Guestrin, 2016), CatBoost (Prokhorenkova et al., 2018), and AdaBoost (Freund and Schapire, 1997). Next, we include 4 representative deep learning models: GRU (Cho et al., 2014), LSTM (Hochreiter and Schmidhuber, 1997), CNN (LeCun et al., 2002), and Transformer (Vaswani et al., 2017). We further incorporate two widely used alpha libraries, Alpha-158 (Microsoft, 2025a) and Alpha-360 (Microsoft, 2025b), as baseline factor sets. We also include two closely related automated alpha mining methods, AutoAlpha (Kou et al., 2024) and AlphaAgent (Tang et al., 2025), for comparison. In addition, six LLMs are evaluated to assess their capacity for alpha mining. Among them, two are closed-source models: GPT-4.1 (OpenAI, 2025), a non-

Models		Dataset	Horizon	CSI300					
				IC	RankIC	ICIR	RankICIR	AER	IR
Machine-Learning	Linear	CSI300	10 days	0.0165	0.0211	0.1612	0.1655	-0.0076	-0.0756
	MLP			0.0227	0.0327	0.2227	0.3037	0.0678	0.9351
	RandomForest			0.0240	0.0410	0.2932	0.4385	0.0784	0.8381
	LightGBM			0.0269	0.0412	0.2811	0.3327	0.0878	1.0980
	XGBoost			0.0257	0.0376	0.2783	0.4093	0.1081	1.3166
	CatBoost			0.0197	0.0239	0.2196	0.3043	0.0462	0.5373
	Adaboost			0.0187	0.0284	0.2709	0.3369	0.1138	1.2633
Deep-Learning	Transformer	CSI300	10 days	-0.0090	-0.0022	-0.0800	-0.0181	0.0492	0.6361
	GRU			0.0074	0.0176	0.0747	0.1370	0.0335	0.3386
	LSTM			0.0096	0.0216	0.0886	0.1619	0.0593	0.6030
	CNN			0.0268	0.0392	0.2432	0.3117	0.0763	0.9642
Libraries and Methods	Alpha 158	CSI300	10 days	0.0358	0.0402	0.2737	0.2866	0.0946	0.8556
	Alpha 360			0.0200	0.0136	0.1674	0.1067	0.1198	1.0762
	AutoAlpha			0.0211	0.0177	0.2030	0.1588	0.0658	0.6307
	AlphaAgent			0.0246	0.0289	0.2407	0.2721	0.1072	1.2310
LLM	Llama3 8B	CSI300	10 days	0.0121	-0.0074	0.0972	-0.0540	0.0520	0.5077
	Llama3 70B			0.0205	0.0229	0.1786	0.1915	0.0681	0.6312
	gpt-oss-20B			0.0061	0.0075	0.0613	0.0680	0.0464	0.4885
	gpt-oss-120B			0.0300	0.0318	0.2501	0.2595	0.0789	0.8015
	GPT-4.1			0.0118	0.0114	0.1069	0.1037	0.0360	0.3628
	o3			0.0019	-0.0050	0.0203	-0.0475	0.0218	0.2278
CogAlpha	gpt-oss-120B	CSI300	10 days	**0.0591**	**0.0814**	**0.3410**	**0.4350**	**0.1639**	**1.8999**

Table 1: Performance comparison between CogAlpha and 21 baseline methods on the CSI 300 constituent stock dataset. The best performance values for each task are highlighted in **bold**.

reasoning model, and o3 (OpenAI, 2025b), a reasoning model. The remaining four are open-source models of different scales and origins: Llama3-8B, Llama3-70B (Grattafiori et al., 2024), GPT-OSS-20B, and GPT-OSS-120B (OpenAI, 2025a). For methods and LLMs that generate alpha factors, we evaluate performance using multi-factor combinations constructed from the 20 generated alphas.

As shown in Table 1, traditional machine learning methods generally exhibit better overall performance than deep learning methods. There is no substantial performance gap between traditional machine learning models and existing alpha libraries or LLM-driven methods that mine formulaic alphas. Moreover, from the four predictive metrics of Alpha158 and Alpha360, we observe that a larger number of alpha factors does not necessarily lead to higher IC or RankIC values. For open-source LLMs, larger models generally exhibit stronger alpha-mining capabilities than smaller ones. Surprisingly, the two closed-source models perform poorly, with the reasoning-oriented model achieving the worst performance among all evaluated LLMs. Overall, CogAlpha consistently outperforms all baseline methods, achieving superior results across all evaluation metrics.

4.3 Ablation Study

In this section, we evaluate the effectiveness of each component of CogAlapha: Adaptive Generation (**A**), Diversified Guidance (**G**), Seven-Level Agent Hierarchy (**H**), and Thinking Evolution (**E**). As shown in Table 3 in Appendix B.5, the four parts can, to some extent, improve the effectiveness and performance of alpha mining.

4.4 Interpretability of Generated Alpha

In this section, we analyze the interpretability of the generated alphas. Each alpha produced by **CogAlpha** is accompanied by detailed comments that explain its logic, clarify its underlying idea, and present the corresponding formula. Following the comments, the implementation code is provided.

The Python code in Listing 1 is an example of a generated alpha. It measures the **liquidity impact**: the price rise (high – close) per unit of traded volume.

$$\text{Alpha} = \frac{day_{high} - day_{close}}{day_{volume} + \varepsilon}. \quad (1)$$

A large positive value indicates that the stock price increased sharply while trading volume remained low, implying thin liquidity and a higher expected short-term return. This design can be interpreted as a measure of the *price impact per unit of traded volume*. In market microstructure theory, this reflects the liquidity constraint between price movements and trading volume: large price changes under low volume often signal poor liquidity, an imbalanced order book, and markets where small trades can

move prices significantly. Such conditions may imply short-term reversal or momentum effects, consistent with the findings of *Continuous Auctions and Insider Trading* (Kyle, 1985) and *Illiquidity and Stock Returns* (Amihud, 2002) on price impact and illiquidity-return relationships.

Listing 1: Initial alpha measuring liquidity impact

```
def factor_upward_impact_per_vol(df):
    """Liquidity-impact: price rise (high-close) per unit of
          traded volume. A large positive value means the
          stock moved up strongly while volume stayed low,
          indicating thin liquidity and higher expected short
          -term return.
    Formula: (high - close) / (volume + ε)."""
    df_copy = df.copy()
    eps = 1e-9
    df_copy['price_up'] = df_copy['high'] - df_copy['close']
    df_copy['factor_upward_impact_per_vol'] = df_copy['
          price_up'] / (df_copy['volume'] + eps)
    return df_copy['factor_upward_impact_per_vol']
```

4.5 Evolution of Alphas

To demonstrate the evolution capability of **CogAlpha**, we show an example of how liquidity-related alphas evolve over multiple iterations. Each generated alpha is evaluated by predictive metrics (IC and RankIC). Poorly performing alphas are automatically filtered out, while stronger ones are preserved and further evolved.

The first version (Listing 1) represents an initial manually designed alpha. It measures the liquidity impact as the price rise $(high - close)$ per unit of traded volume. Its metrics are: **IC: 0.0090**, **RankIC: 0.0061**. Through mutation, the model generates an alternative formulation (Listing 2 in Appendix B.6) that uses the full daily price range $(high - low)$ instead of the closing difference. This captures broader intraday liquidity behavior. Its metrics slightly decrease to **IC: 0.0073**, **RankIC: 0.0021**, and thus this version is discarded in later rounds. After several evolutionary rounds, CogAlpha produces a more refined version (Listing 3 in Appendix B.6). It normalizes the absolute daily price move by dollar volume and applies a $\tanh$ transformation to ensure boundedness and robustness. The evolved alpha achieves significantly improved performance with **IC: 0.0141** and **RankIC: 0.0087**, showing the ability of the evolutionary mechanism to refine quantitative factors effectively.

After a complete evolution cycle, CogAlpha is able to generate a large number of single-factor alphas with strong predictive power, many of which achieve **absolute IC values above 0.05** and **absolute RankIC values above 0.07**. This demonstrates the framework's capacity to autonomously explore and optimize factor space toward higher-performing and more interpretable alphas.

4.6 Generalization to Different Settings

To test the generalization of CogAlpha, we conduct experiments on five different datasets (CSI300, CSI500, S&P500, HSI, HSCI) from three different stock markets (China, U.S., and HK), using two training methods (LightGBM and Ridge) and two prediction horizons (10 days and 30 days). As shown in Table 4 in Appendix B.8, our method consistently performs well across different settings.

4.7 Different Fitness Threshold

In this section, we analyze the sensitivity of our method to different threshold settings used for filtering alpha factors. To maintain the quality of the selected alphas, we experiment with three threshold pairs: (65, 80), (80, 90), and (85, 95). In each pair, the former value represents the percentile threshold for *qualified factors* that advance to the next generation, while the latter corresponds to the percentile threshold for *elite factors* that are directly stored in the final candidate pool. For comparison, we also establish a baseline configuration to ensure the quality consistency of filtered alphas. Specifically, thresholds for each predictive metric are determined based on the empirical distribution of factor scores. We constrain each metric by a minimum bound to prevent the dominance of outliers: *IC* and *RankIC* are bounded below by 0.005, *ICIR* and *RankICIR* by 0.05, and *MI* by 0.02. For elite factors, the minimum bounds are slightly higher: 0.01 for *IC* and *RankIC*, 0.1 for *ICIR* and *RankICIR*, and 0.02 for *MI*. As shown in Figure 3 in Appendix B.7, the threshold pair (65, 80) yields better overall performance. This result may be attributed to the larger parent pool size under this configuration, which encourages evolutionary search to explore a broader alpha space and mitigates the risk of premature convergence to local optima.

5 Conclusion

In this work, we study how to extract interpretable and reliable alpha signals from financial markets characterized by high volatility and a low signal-to-noise ratio. We introduce the concept of *Cognitive Alpha Mining*, which opens a new direction for automated, robust, and explainable alpha discovery. We further propose CogAlpha, a multi-agent framework powered by deeper-thinking LLMs. Extensive experiments demonstrate the effectiveness of our approach. In future work, we plan to implement our method in live trading environments to further validate its practical performance.

Limitations

The CogAlpha framework is intended for academic use only and does not provide any financial opinions. The backtesting simulations are implemented and executed entirely within the Qlib framework, which may not fully replicate the conditions of live trading environments. Additionally, due to the inherent randomness of LLM outputs, reproducing exactly the same alphas in each run can be challenging. Furthermore, the execution time of the experiments is influenced by the size of the dataset, with larger datasets potentially leading to longer processing times.

Ethical Considerations

All datasets used in this paper were downloaded from public sources and are publicly available.

We used OpenAI's ChatGPT-5.2 for grammar checking and suggestions, but manually verified all edits. No AI-generated content was directly included in the final submission.

Users of the CogAlpha framework and its related code are responsible for sourcing their own financial data and independently evaluating the risks associated with the generated factors and models in their specific contexts. It is crucial to approach the agent-generated code, data, and models with care and perform comprehensive verification. The CogAlpha framework does not offer financial advice and is not intended to substitute the expertise of qualified financial professionals in the creation, evaluation, and approval of financial products.

References

AASTocks. 2025. Hong kong stock market index – hsi and others. https://www.aastocks.com/en/stocks/market/index/hk-index-con.aspx. Accessed: 2025-12-05.

Yakov Amihud. 2002. Illiquidity and stock returns: cross-section and time-series effects. *Journal of financial markets*, 5(1):31–56.

Ran Aroussi. 2024. yfinance: Download market data from yahoo! finance's api. https://pypi.org/project/yfinance/. Accessed: 2025-12-05.

Leo Breiman. 2001. Random forests. *Machine learning*, 45(1):5–32.

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Can Cui, Wei Wang, Meihui Zhang, Gang Chen, Zhaojing Luo, and Beng Chin Ooi. 2021. Alphaevolve: A learning framework to discover novel alphas in quantitative investment. In *Proceedings of the 2021 International conference on management of data*, pages 2208–2216.

Yitong Duan, Lei Wang, Qizhong Zhang, and Jian Li. 2022. Factorvae: A probabilistic dynamic factor model based on variational autoencoder for predicting cross-sectional stock returns. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pages 4468–4476.

Robert F Engle. 1982. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica: Journal of the econometric society*, pages 987–1007.

Eugene F Fama and Kenneth R French. 1992. The cross-section of expected stock returns. *the Journal of Finance*, 47(2):427–465.

Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jian Guo, Saizhuo Wang, Lionel M Ni, and Heung-Yeung Shum. 2024a. Quant 4.0: engineering quantitative investment with automated, explainable, and knowledge-driven artificial intelligence. *Frontiers of Information Technology & Electronic Engineering*, 25(11):1421–1445.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024b. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*.

Hang Seng Indexes Company. 2025. Hang seng composite index – all indexes. https://www.hsi.com.hk/eng/indexes/all-indexes/hsci?from=companyhomepages.com. Accessed: 2025-12-05.

Campbell R Harvey, Yan Liu, and Heqing Zhu. 2016. … and the cross-section of expected returns. *The Review of Financial Studies*, 29(1):5–68.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. 2024. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Kewei Hou, Chen Xue, and Lu Zhang. 2017. Replicating anomalies. Technical report, National Bureau of Economic Research.

Vedant Dhaval Jobanputra, Basam Thilaknath Reddy, Sri Ganesh Bhojanapalli, Krishna Aditya SV S, Bagavathi Chandrasekara, and Ritwik Murali. 2025. Llm-aided evolutionary algorithms for haiku generation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2584–2587.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30.

Zhizhuo Kou, Holam Yu, Junyu Luo, Jingshu Peng, Xujia Li, Chengzhong Liu, Juntao Dai, Lei Chen, Sirui Han, and Yike Guo. 2024. Automate strategy finding with llm in quant investment. *arXiv preprint arXiv:2409.06289*.

Albert S Kyle. 1985. Continuous auctions and insider trading. *Econometrica: Journal of the Econometric Society*, pages 1315–1335.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 2002. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. 2025. Evolving deeper llm thinking. *arXiv preprint arXiv:2501.09891*.

Yuante Li, Xu Yang, Xiao Yang, Minrui Xu, Xisen Wang, Weiqing Liu, and Jiang Bian. 2025. R&d-agent-quant: A multi-agent framework for data-centric factors and model joint optimization. *arXiv preprint arXiv:2505.15155*.

Xiaoming Lin, Ye Chen, Ziyu Li, and Kang He. 2019. Stock alpha mining based on genetic algorithm. *Technical Report, Huatai Securities Research Center*.

Xiao-Yang Liu, Hongyang Yang, Jiechao Gao, and Christina Dan Wang. 2021. Finrl: Deep reinforcement learning framework to automate trading in quantitative finance. In *Proceedings of the second ACM international conference on AI in finance*, pages 1–9.

Microsoft. 2025a. Alpha 158 from microsoft qlib. https://github.com/microsoft/qlib/blob/85cc74846b5af2e3e6d18666a2f6e399396980b9/qlib/contrib/data/loader.py#L61. Accessed: 2025-05-12.

Microsoft. 2025b. Alpha 360 from microsoft qlib. https://github.com/microsoft/qlib/blob/85cc74846b5af2e3e6d18666a2f6e399396980b9/qlib/contrib/data/loader.py#L4. Accessed: 2025-05-12.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, and 1 others. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*.

OpenAI. 2025a. GPT-OSS-120B & GPT-OSS-20B model card. *arXiv preprint arXiv:2508.10925*.

OpenAI. 2025b. Introducing deep research. https://openai.com/index/introducing-deep-research/. Accessed: 2025-09-24.

OpenAI. 2025. Introducing GPT-4.1 in the api. https://openai.com/index/gpt-4-1/. Accessed: 2025-11-11.

Rahul Ramesh Patil. 2023. Ai-infused algorithmic trading: Genetic algorithms and machine learning in high-frequency trading. *International Journal For Multidisciplinary Research*, 5(5).

Giovanni Pinna, Damiano Ravalico, Luigi Rovito, Luca Manzoni, and Andrea De Lorenzo. 2024. Enhancing large language models-based code generation by leveraging genetic improvement. In *European Conference on Genetic Programming (Part of EvoStar)*, pages 108–124. Springer.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, and 1 others. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Michael D Schmidt and Hod Lipson. 2010. Age-fitness pareto optimization. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 543–544.

Hao Shi, Weili Song, Xinting Zhang, Jiahe Shi, Cuicui Luo, Xiang Ao, Hamid Arian, and Luis Angel Seco. 2025a. Alphaforge: A framework to mine and dynamically combine formulaic alpha factors. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 12524–12532.

Yu Shi, Yitong Duan, and Jian Li. 2025b. Navigating the alpha jungle: An llm-powered mcts framework for formulaic factor mining. *arXiv preprint arXiv:2505.11122*.

Zhaofan Su, Jianwu Lin, and Chengshan Zhang. 2022. Genetic algorithm based quantitative factors construction. In *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*, pages 650–655. IEEE.

Ziyi Tang, Zechuan Chen, Jiarui Yang, Jiayao Mai, Yongsen Zheng, Keze Wang, Jinrui Chen, and Liang Lin. 2025. Alphaagent: Llm-driven alpha mining with regularized exploration to counteract alpha decay. *arXiv preprint arXiv:2502.16789*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Le Xiao and Xiaolin Chen. 2023. Enhancing llm with evolutionary fine tuning for news summary generation. *arXiv preprint arXiv:2307.02839*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Wentao Xu, Weiqing Liu, Lewen Wang, Yingce Xia, Jiang Bian, Jian Yin, and Tie-Yan Liu. 2021a. Hist: A graph-based framework for stock trend forecasting via mining concept-oriented shared information. *arXiv preprint arXiv:2110.13716*.

Wentao Xu, Weiqing Liu, Chang Xu, Jiang Bian, Jian Yin, and Tie-Yan Liu. 2021b. Rest: Relational event-driven stock trend forecasting. In *Proceedings of the web conference 2021*, pages 1–10.

Xiao Yang, Weiqing Liu, Dong Zhou, Jiang Bian, and Tie-Yan Liu. 2020. Qlib: An ai-oriented quantitative investment platform. *arXiv preprint arXiv:2009.11189*.

Shuo Yu, Hongyan Xue, Xiang Ao, Feiyang Pan, Jia He, Dandan Tu, and Qing He. 2023. Generating synergistic formulaic alpha collections via reinforcement learning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5476–5486.

Tianping Zhang, Yuanqi Li, Yifei Jin, and Jian Li. 2020. Autoalpha: an efficient hierarchical evolutionary algorithm for mining alpha factors in quantitative investment. *arXiv preprint arXiv:2002.08245*.

Tianping Zhang, Zheyu Aqa Zhang, Zhiyuan Fan, Haoyan Luo, Fengyuan Liu, Qian Liu, Wei Cao, and Li Jian. 2023. Openfe: Automated feature generation with expert-level performance. In *International Conference on Machine Learning*, pages 41880–41901. PMLR.

A Approach

A.1 Seven-Level Agent Hierarchy

- **Level 1: Market Structure & Cycle Layer** *AgentMarketCycle* explores long-term cyclical transitions and phase shifts in price dynamics, revealing hidden market rhythms and structural turning points. *AgentVolatilityRegime* detects transitions between calm and turbulent volatility states, characterizing regime persistence and clustering behavior.

- **Level 2: Extreme Risk & Fragility Layer** *AgentTailRisk* quantifies downside sensitivity and tail-event exposure, modeling how negative shocks propagate through time. *AgentCrashPredictor* identifies early warning signals of market collapses by tracking volatility compression, liquidity depletion, and structural fragility patterns.

- **Level 3: Price–Volume Dynamics Layer** *AgentLiquidity* measures market depth and trading frictions through price impact and turnover variability. *AgentOrderImbalance* captures directional pressure from one-sided participation inferred from daily OHLCV patterns. *AgentPriceVolumeCoherence* examines synchronization and divergence between price and volume changes, revealing energy alignment or decoupling. *AgentVolumeStructure* analyzes the statistical shape and concentration of trading activity to understand participation rhythm and clustering.

- **Level 4: Price–Volatility Behavior Layer** *AgentDailyTrend* models directional persistence and multi-day momentum strength to uncover sustained price movements. *AgentReversal* captures mean-reversion and short-term overreaction corrections following transient mispricings. *AgentRangeVol* investigates range-based volatility dynamics, including compression–expansion cycles in daily price ranges. *AgentLagResponse* studies delayed price adjustments and lagged feedback between volatility, volume, and returns. *AgentVolAsymmetry* measures asymmetric volatility between upward and downward price moves, highlighting skewed risk behavior.

- **Level 5: Multi-Scale Complexity Layer** *AgentDrawdown* evaluates the depth, duration,

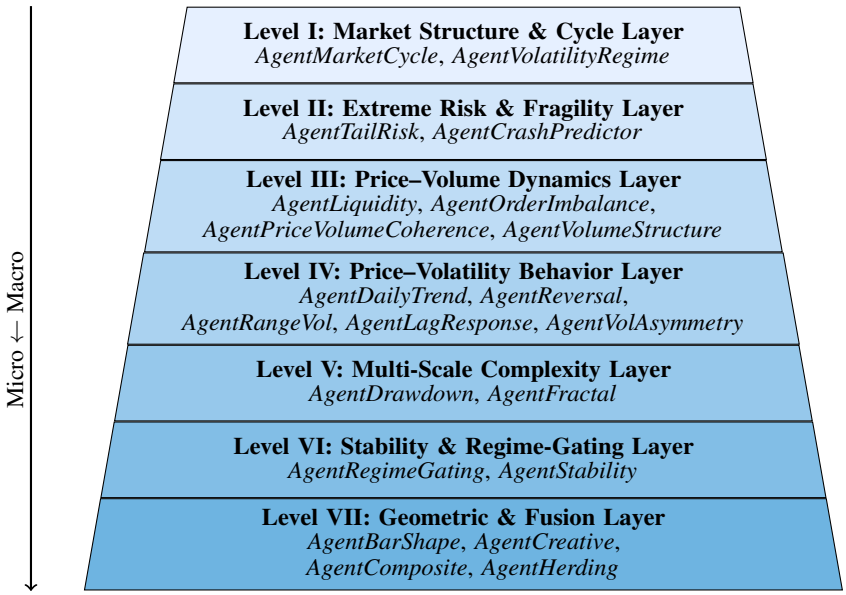

Figure 2: **Seven-Level Agent Hierarchy** (Top–Down Pyramid). The pyramid illustrates the seven-level agent hierarchy from macro-structural reasoning to micro-level fusion.

and recovery geometry of cumulative losses, emphasizing temporal resilience. *AgentFractal* assesses multi-scale roughness and long-memory characteristics through cross-horizon variability and structural irregularity.

- **Level 6: Stability & Regime-Gating Layer** *AgentRegimeGating* constructs adaptive gates that modulate signal activation depending on volatility, trend, or liquidity states. *AgentStability* quantifies temporal consistency and persistence in returns or derived signals, emphasizing robustness and smoothness.

- **Level 7: Geometric & Fusion Layer** *AgentComposite* fuses multiple independent factors into coherent composites, emphasizing synergy and orthogonality among signals. *AgentCreative* applies non-linear transformations, reparametrizations, or soft gating to generate novel feature representations. *AgentBarShape* encodes candlestick geometry—body, shadow, and symmetry—into continuous and interpretable quantitative descriptors. *AgentHerding* detects collective crowding behavior and directional alignment within OHLCV dynamics, reflecting market consensus intensity.

A.2 Diversified Guidance

- **Light**: Performs minimal rewording to maintain nearly identical meaning while improving clarity and linguistic fluency. It serves as a baseline for consistency testing across linguistic variations.

- **Moderate**: Rephrases the content naturally with mild enrichment or stylistic variation. This helps capture nuanced semantic differences and tests factor robustness under slightly altered descriptive framing.

- **Creative**: Introduces expressive, research-oriented rewording that adds interpretative depth. This style aims to inspire novel analytical angles or alternative reasoning patterns that remain aligned with the original domain.

- **Divergent**: Produces exploratory rewrites from new but relevant analytical viewpoints, often shifting emphasis toward different sub-mechanisms within the same conceptual framework. This encourages broader hypothesis generation and factor diversity.

- **Concrete**: Makes the guidance more specific and implementation-oriented by introducing measurable quantities such as statistical formulas, ratios, or example computations. This version bridges conceptual factor ideas with practical implementation cues.

A.3 Multi-Agent Quality Checker

As shown in Figure 1, the Multi-Agent Quality Checker operates through the following sequence:

Level	Layer Name	Description
I	Market Structure & Cycle Layer	Explores large-scale temporal structures such as long-term trends, market phases, and cyclical state transitions inferred from daily OHLCV dynamics.
II	Extreme Risk & Fragility Layer	Models tail-risk exposure, crash precursors, and systemic fragility patterns that signal potential regime breakdowns or stress accumulation.
III	Price–Volume Dynamics Layer	Captures the interactions between price and trading activity—liquidity, order imbalance, and coherence between price movement and volume behavior.
IV	Price–Volatility Behavior Layer	Analyzes trend persistence, short-term reversals, volatility clustering, and asymmetric price dynamics as core sources of predictive alpha.
V	Multi-Scale Complexity Layer	Measures cross-scale irregularities, fractal roughness, drawdown–recovery geometry, and long-memory characteristics in time-series structures.
VI	Stability & Regime-Gating Layer	Assesses temporal stability and constructs adaptive gating mechanisms that regulate signal activation under varying market conditions.
VII	Geometric & Fusion Layer	Focuses on geometric pattern representation (candlestick morphology) and multi-factor fusion, combining independent signals into coherent composite factors.

Table 2: Seven-Level Agent Hierarchy of CogAlpha and their corresponding conceptual focuses.

Code Quality Agent. It performs the first-pass audit of the raw LLM-generated code. It detects syntactic errors, undefined variables, formatting inconsistencies, invalid library calls, and potential runtime failures using static analysis and lightweight interpreter checks. This agent ensures that the code is structurally well-formed before deeper semantic inspection takes place.

Code Repair Agent. If the Code Quality Agent identifies issues, the Code Repair Agent attempts to fix them autonomously. Repairs include correcting import statements, rewriting malformed expressions, resolving type mismatches, and rewriting unstable numerical operations. This agent ensures that the factor is at least syntactically and operationally viable.

Judge Agent. Once the code is syntactically clean, the Judge Agent evaluates the factor at a semantic level. It assesses whether the factor is:

- *Logically consistent*: correct operator ordering, coherent data flow, no degenerate expressions;
- *Technically correct*: valid use of rolling windows, transforms, and TA-Lib functions;
- *Economically meaningful*: obeys financial intuition and avoids fabricated indicators.

Factors that fail this semantic audit are routed to the Logic Improvement Agent.

Logic Improvement Agent. This agent refines factors that exhibit weak or inconsistent logic. It restructures formulas, adjusts window parameters, replaces dubious transformations, eliminates redundant operations, and enhances the overall financial interpretability while preserving the original modeling intent. This refinement improves robustness without altering the factor's core hypothesis.

Execution and Numerical Stability Check. After passing the logical audits, the code is executed in a restricted sandbox. We evaluate numerical stability by detecting runtime errors, *NaN* propagation, overflow/underflow, invalid logarithms, and unstable normalizations. Codes that fail are rejected or sent back to earlier agents for correction.

Temporal Leakage Unit Test. Finally, Static Safety conducts a domain-specific leakage test to ensure that the factor does not use future information. This test detects forward-looking shifts (e.g., `shift(-1)`), misaligned rolling windows, or implicit temporal violations that may pass standard code-safety tools. Only factors with zero leakage are accepted.

Output. Codes that pass all agents form a pool of safe, executable, and leakage-free alpha factors. These factors constitute the foundation for the next stage, *Thinking Evolution*, which focuses on improving reasoning reliability and logical effectiveness. By enforcing strict correctness at the code level first, CogAlpha ensures that all evolution occurs on top of a solid and trustworthy computational base.

A.4 Fitness Evaluation

The threshold values for the five predictive power metrics may vary depending on the dataset. For example, on the CSI300, we constrain each metric with a minimum bound to prevent the dominance of outliers: *IC* and *RankIC* are bounded below by 0.005, *ICIR* and *RankICIR* by 0.05, and *MI* by 0.02. For elite factors, the minimum bounds are slightly higher: 0.01 for *IC* and *RankIC*, 0.1 for *ICIR* and *RankICIR*, and 0.02 for *MI*. However, on the S&P500, we apply similar constraints to prevent the dominance of outliers: *IC* and *RankIC* are bounded below by 0.005, *ICIR* and *RankICIR* by 0.05, and *MI* by 0.012. For elite factors, the minimum bounds are slightly higher: 0.01 for *IC* and *RankIC*, 0.1 for *ICIR* and *RankICIR*, and 0.012 for *MI*. This is because it is harder to mine alpha signals in a more effective stock market.

Non-linear relationships (e.g., MI) suggest that the market may not be fully efficient, and certain information may not be fully reflected in prices, thus providing opportunities for factor investing. Non-linear factor models are better able to capture complex patterns in the market and may offer opportunities for excess returns, especially when the market is not fully efficient.

B Experiments

B.1 Datasets

Our experiments are mainly conducted on the CSI300 (China Securities Index 300). Its components consist of 300 large-cap A-share stocks in the Chinese market. We primarily use the 10-day return as the prediction target, with buying and selling at the open price. The dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2024/12/01) periods. On the same dataset, we also use a 30-day return as the prediction target.

CSI500 (China Securities Index 500) consists of 500 relatively smaller but liquid companies in China. We primarily use the 10-day return as the prediction target, with buying and selling at the open price. The dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2024/12/01) periods.

Three other datasets from two different stock markets (U.S. and HK) are also used. The S&P500 is the Standard & Poor's 500 Index, and its components include 500 of the largest publicly traded companies in the U.S. stock market. The dataset is split chronologically into training (2007/01/01-2014/12/31), validation (2015/01/01-2015/12/31), and test (2016/01/01-2020/12/01) periods.

The HSI (Hang Seng Index) tracks the performance of the largest and most liquid companies listed on the Hong Kong Stock Exchange. Currently, according to (AASTocks, 2025), it has 89 stocks. The dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2025/12/01) periods.

The HSCI (Hang Seng China Enterprises Index) covers about the top 95th percentile of the total market capitalization of companies listed on the Main Board of the Stock Exchange of Hong Kong. Currently, according to (Hang Seng Indexes Company, 2025), it includes 509 stocks. The dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2025/12/01) periods.

The CSI300, CSI500, and S&P500 datasets are downloaded from the Qlib platform (Yang et al., 2020). The HSI and HSCI datasets are downloaded from Yahoo Finance (Aroussi, 2024). All backtesting simulations are implemented and executed entirely within the Qlib framework.

B.2 Backtest

The top-50/drop-5 strategy is a ranking-based portfolio construction method that selects the top 50 stocks with the highest predicted returns while limiting daily portfolio turnover. On each trading

day, the portfolio retains previously selected high-ranking stocks and replaces at most 5 positions. All trades are executed at the opening price. The open cost is set to 0.05%, and the close cost is set to 0.15%. A minimum transaction fee of 5 CNY is applied to each trade.

B.3 Metrics

We use five factor predictive power metrics: Information Coefficient (**IC**), Information Coefficient Information Ratio (**ICIR**), Rank Information Coefficient (**RankIC**), Rank Information Coefficient Information Ratio (**RankICIR**), and Mutual Information (**MI**). Assume there are N_t assets at time t. Let $f_{i,t}$ represent the predicted returns for asset i at time t, and r_{t+1} represent the total return over the subsequent period, from t to $t+1$. The evaluation spans over T time periods. We also use two performance metrics: AER and IR.

The **Information Coefficient (IC)** measures the linear correlation between factor values and subsequent total returns. It is the average of each linear cross-sectional relationship between factor values and subsequent returns at time t over all T periods:

$$\begin{aligned} \mathrm{IC} &= \frac{1}{T}\sum_{t=1}^{T}\mathrm{IC}_t \\ \mathrm{IC}_t &= \frac{\sum_{i=1}^{N_t}(f_{i,t}-\bar{f}_t)(r_{i,t+1}-\bar{r}_{t+1})}{\sqrt{\sum_{i=1}^{N_t}(f_{i,t}-\bar{f}_t)^2}\sqrt{\sum_{i=1}^{N_t}(r_{i,t+1}-\bar{r}_{t+1})^2}} \end{aligned} \tag{2}$$

The **Information Coefficient Information Ratio (ICIR)** evaluates the stability of **IC** across time:

$$\mathrm{ICIR} = \frac{\mathbb{E}[\mathrm{IC}_t]}{\mathrm{Std}[\mathrm{IC}_t]} \approx \frac{\mathrm{IC}}{\mathrm{Std}(\{\mathrm{IC}_t\}_{t=1}^{T})}, \tag{3}$$

where IC denotes the time-averaged IC_t.

The **Rank Information Coefficient (RankIC)** measures the monotonic relationship between the factor and the subsequent total returns. Let

$$u_{i,t} = \mathrm{rank}(f_{i,t}), \qquad v_{i,t} = \mathrm{rank}(r_{i,t+1}),$$

and their means $\bar{u}_t$, $\bar{v}_t$ across N_t assets. Analogous to IC, RankIC over period T can be expressed as:

$$\begin{aligned} \mathrm{RankIC} &= \frac{1}{T}\sum_{t=1}^{T}\mathrm{RankIC}_t \\ \mathrm{RankIC}_t &= \frac{\sum_{i=1}^{N_t}(u_{i,t}-\bar{u}_t)(v_{i,t}-\bar{v}_t)}{\sqrt{\sum_{i=1}^{N_t}(u_{i,t}-\bar{u}_t)^2}\sqrt{\sum_{i=1}^{N_t}(v_{i,t}-\bar{v}_t)^2}}. \end{aligned} \tag{4}$$

Analogous to ICIR, the **RankIC Information Ratio (RankICIR)** measures the temporal stability of RankIC:

$$\mathrm{RankICIR} = \frac{\mathbb{E}[\mathrm{RankIC}_t]}{\mathrm{Std}[\mathrm{RankIC}_t]} \approx \frac{\overline{\mathrm{RankIC}}}{\mathrm{Std}(\{\mathrm{RankIC}_t\}_{t=1}^{T})}. \tag{5}$$

The **Mutual Information (MI)** captures the non-linear dependence between factor values and the subsequent total returns. It measures the reduction in uncertainty of R given knowledge of F:

$$\mathrm{MI}(F,R) = \iint p(f,r)\,\log\frac{p(f,r)}{p(f)\,p(r)}\,df\,dr, \tag{6}$$

where $p(f,r)$ denotes the joint density of factor f and return r, and $p(f)$, $p(r)$ are their respective marginal densities. A higher MI implies a stronger (possibly nonlinear) dependency between the factor and subsequent returns.

Annualized Excess Return (AER). Following Qlib's implementation, we compute daily excess returns as

$$r_t = r_t^{\text{port}} - r_t^{\text{bench}} - \text{cost}_t,$$

where r_t^{port} is the portfolio return, r_t^{bench} is the benchmark return, and cost_t is the transaction cost. The average daily excess return is

$$\mu = \frac{1}{T}\sum_{t=1}^{T} r_t,$$

and the annualized excess return is obtained via arithmetic scaling:

$$\mathrm{AER} = \mu \times N,$$

where N is the number of trading periods in a year (e.g., $N = 252$ for daily returns).

Information Ratio (IR). The standard deviation of daily excess returns is

$$\sigma = \sqrt{\frac{1}{T-1}\sum_{t=1}^{T}(r_t-\mu)^2}.$$

Qlib annualizes the Information Ratio using

$$\mathrm{IR} = \frac{\mu}{\sigma}\sqrt{N}.$$

B.4 Training Setting

The size of the initial pool is set to 80, meaning that the minimum number of alphas generated by the task-specific agents is 80. The parent pool size is set to 32, indicating that after filtering, at most 32

Models	Dataset	Horizon	CSI300					
			IC	RankIC	ICIR	RankICIR	AER	IR
Agent	CSI300	10 days	0.0300	0.0318	0.2501	0.2595	0.0789	0.8015
Agent_E			0.0219	0.0420	0.1932	0.3322	0.0808	0.8999
Agent_EA			0.0315	0.0491	0.2568	0.3583	0.0825	1.0145
Agent_EAG			0.0414	0.0501	0.3239	0.3599	0.1245	1.4668
Agent_EAGH (CogAlpha)			**0.0591**	**0.0814**	**0.3410**	**0.4350**	**0.1639**	**1.8999**

Table 3: Ablation study of CogAlpha. **A** denotes Adaptive Generation, **G** denotes Diversified Guidance, **H** denotes the Seven-Level Agent Hierarchy, and **E** denotes Thinking Evolution. The best performance values for each task are highlighted in **bold**.

alphas are retained and passed to the next generation. The children's pool is set to be three times the size of the parent pool, meaning that the minimum number of alphas generated by the evolutionary agents is 96. By default, each task-specific agent leads a complete evolutionary cycle, which consists of 24 generations and 3 inner sub-cycles, with each sub-cycle comprising 8 generations. Thus, each task-specific agent initiates the evolutionary search 3 times. In addition, every 2 generations, new alphas generated by the task-specific agents are filtered and injected into the parent pool. For each generation, the top two elite alphas from the previous generation are always carried forward to the next. All alphas containing more than 30% NaN values or failing the multi-agent quality checker are discarded. All experiments are conducted on NVIDIA H100 GPUs.

We use rolling training with a rolling step of 126. Two models are employed in this work: the *LGBMRegressor* and *Ridge* models. The *LGBMRegressor* is configured with a learning rate of 0.0001, 32 leaves per tree, a maximum depth of 12, and regularization terms (`reg_alpha` and `reg_lambda`) set to 1.0. The model uses a total of 1000 trees with sampling techniques (feature and bagging fractions set to 0.8) to reduce overfitting. For the *Ridge* model, the regularization strength (`alpha`) is set to 10, which controls the regularization applied to the model to prevent overfitting.

Due to the inherent randomness in the outputs of large models, the results may vary with each run. However, factor mining is different from other experiments in that good factors can be accumulated and stored. Therefore, the experimental results presented in this paper reflect the outcomes after a single round of factor mining.

B.5 Ablation Study

We evaluate the effectiveness of each component of CogAlapha: Adaptive Generation (**A**), Diversified Guidance (**G**), Seven-Level Agent Hierarchy (**H**), and Thinking Evolution (**E**). As shown in Table 3, the four parts can, to some extent, improve the effectiveness and performance of alpha mining.

B.6 Evolution of Alphas

To demonstrate the evolution capability of **CogAlpha**, we show an example of how liquidity-related alphas evolve over multiple iterations. Each generated alpha is evaluated by predictive metrics (IC and RankIC). Poorly performing alphas are automatically filtered out, while stronger ones are preserved and further evolved.

Listing 2: Mutated alpha variant using full price range

```
def factor_dayhigh_impact_per_vol(df):
    """Price-impact proxy: (high-low) per unit of volume.
    Larger values indicate that price moves a lot while
        little volume trades, signalling thin liquidity."""
    df_copy = df.copy()
    df_copy['price_range'] = df_copy['high'] - df_copy['low'
        ]
    df_copy['factor_dayhigh_impact_per_vol'] = df_copy['
        price_range'] / (df_copy['volume'] + 1e-9)
    return df_copy['factor_dayhigh_impact_per_vol']
```

The first version (Listing 1) represents an initial manually designed alpha. It measures the liquidity impact as the price rise $(high - close)$ per unit of traded volume. Its metrics are: **IC: 0.0090**, **RankIC: 0.0061**. Through mutation, the model generates an alternative formulation (Listing 2) that uses the full daily price range $(high - low)$ instead of the closing difference. This captures broader intraday liquidity behavior. Its metrics slightly decrease to **IC: 0.0073**, **RankIC: 0.0021**, and thus this version is discarded in later rounds.

Listing 3: Evolved alpha after multi-round optimization

```
def factor_price_impact_per_vol_tanh_1d(df):
    """Impact proxy: absolute daily price move per dollar
        volume.
    Steps:
    1) Compute absolute price move (|Close-Open|).
    2) Compute dollar volume (Volume*Close).
    3) Form raw impact = absolute move / (dollar volume + ε)
        .
    4) Apply tanh to bound the factor within (-1, 1)."""
    df_copy = df.copy()
    eps = 1e-9
    df_copy.loc[:, "abs_move"] = (df_copy["close"] - df_copy
        ["open"]).abs()
```

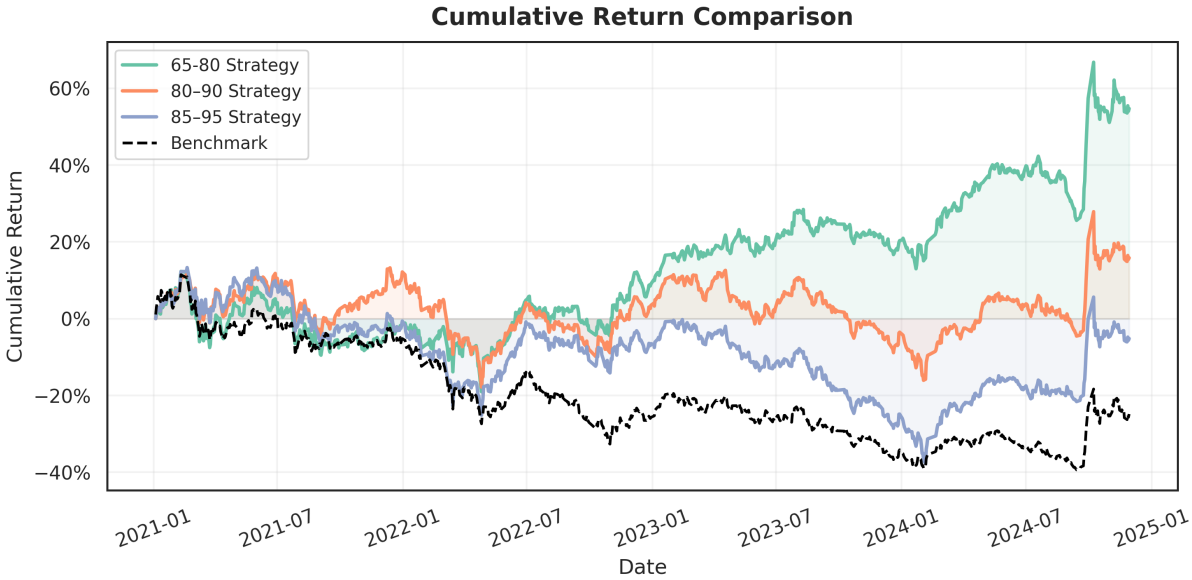

Figure 3: **Performance of CogAlpha on different fitness threshold**

```
    df_copy.loc[:, "dollar_vol"] = df_copy["volume"] *
        df_copy["close"]
    df_copy.loc[:, "raw_impact"] = df_copy["abs_move"] / (
        df_copy["dollar_vol"] + eps)
    df_copy.loc[:, "factor_price_impact_per_vol_tanh_1d"] =
        np.tanh(df_copy["raw_impact"])
    return df_copy["factor_price_impact_per_vol_tanh_1d"]
```

After several evolutionary rounds, CogAlpha produces a more refined version (Listing 3). It normalizes the absolute daily price move by dollar volume and applies a tanh transformation to ensure boundedness and robustness. The evolved alpha achieves significantly improved performance with **IC: 0.0141** and **RankIC: 0.0087**, showing the ability of the evolutionary mechanism to refine quantitative factors effectively.

After a complete evolution cycle, CogAlpha is able to generate a large number of single-factor alphas with strong predictive power, many of which achieve **absolute IC values above 0.05** and **absolute RankIC values above 0.07**. This demonstrates the framework's capacity to autonomously explore and optimize factor space toward higher-performing and more interpretable alphas. The following are a few examples of elite alphas:

factor_lownorm_slopecos_30d_low : *Train metrics (raw)*: IC = -0.0498, RankIC = -0.0791, ICIR = -0.3416, RankICIR = -0.5016; *Test metrics (LightGBM)*: IC = 0.0507, RankIC = 0.0704, ICIR = 0.3116, RankICIR = 0.4262

Listing 4: Evolved alpha after multi-round optimization

```
def factor_lownorm_slopecos_30d_low(df):
    """Low-price relative to its 30-day EMA multiplied by a
        cycle-aligned slope*cosine.
    Steps:
    1. Normalise the low price by its 30-day EMA -> dynamic
        support level.
    2. Estimate the soft-sign slope of log-price vs. log-
        volume (EWMA cov/var) -> short-term trend.
    3. Compute the EMA-12/-48 cosine to capture the dominant
         market cycle phase.
    4. Combine slope and cosine (slope*cosine) as a single "
        cycle factor".
    5. Multiply the normalised low by the cycle factor; the
        product is high when price is near support and the
        cycle direction is favourable."""
    df_copy = df.copy()
    eps = 1e-9

    # 1. Normalised low price (support level)
    low_ema30 = df_copy['low'].ewm(span=30, adjust=False).
        mean()
    norm_low = df_copy['low'] / (low_ema30 + eps)

    # 2. Soft-sign slope of log-price vs. log-volume
    log_ret = np.log(df_copy['close'] / df_copy['close'].
        shift(1))
    log_vol = np.log(df_copy['volume'] / df_copy['volume'].
        shift(1))
    cov = log_ret.ewm(halflife=20, adjust=False).cov(log_vol
        )
    var = log_vol.ewm(halflife=20, adjust=False).var() + eps
    slope_raw = cov / var
    slope = slope_raw / (1.0 + slope_raw.abs())

    # 3. EMA-12/-48 cosine (cycle phase)
    ema12 = df_copy['close'].ewm(span=12, adjust=False).mean
        ()
    ema48 = df_copy['close'].ewm(span=48, adjust=False).mean
        ()
    cos_phase = ema12 / np.sqrt(ema12**2 + ema48**2 + eps)

    # 4. Cycle factor = slope * cosine
    cycle_factor = slope * cos_phase

    # 5. Final factor
    df_copy['factor_lownorm_slopecos_30d_low'] = norm_low *
        cycle_factor

    return df_copy['factor_lownorm_slopecos_30d_low']
```

factor_pressure_drawdown_fisher_10d : *Train metrics (raw)*: IC = -0.0473, RankIC = -0.0668, ICIR = -0.3749, RankICIR = -0.4473; *Test metrics (LightGBM)*: IC = 0.0491, RankIC = 0.069, ICIR = 0.2717, RankICIR = 0.3604

Listing 5: Evolved alpha after multi-round optimization

```
def factor_pressure_drawdown_fisher_10d(df):
```

Dataset	Horizon	Models	Training Method	IC	RankIC	ICIR	RankICIR
CSI300	10 days	CNN	CNN	0.0268	0.0392	0.2432	0.3117
		Linear	Linear	0.0165	0.0211	0.1612	0.1655
		XGBoost	XGBoost	0.0257	0.0376	0.2783	0.4093
		CogAlpha	Ridge	0.0539	0.0714	**0.3471**	**0.4380**
		CogAlpha	LightGBM	**0.0591**	**0.0814**	0.3410	0.4350
	30 days	CNN	CNN	0.0445	0.0644	0.4118	0.6023
		Linear	Linear	0.0222	0.0482	0.2174	0.4218
		XGBoost	XGBoost	0.0402	0.0507	0.4181	0.5896
		CogAlpha	Ridge	0.0815	0.1069	0.4408	0.5403
		CogAlpha	LightGBM	**0.0886**	**0.1243**	**0.4933**	**0.6740**
CSI500	10 days	CNN	CNN	0.0353	0.0490	0.2615	0.3959
		Linear	Linear	0.0273	0.0391	0.2787	0.3671
		XGBoost	XGBoost	0.0282	0.0290	0.2458	0.3046
		CogAlpha	Ridge	**0.0455**	**0.0738**	**0.2903**	**0.4752**
S&P500	10 days	CNN	CNN	-0.0040	-0.0028	-0.0508	-0.0325
		Linear	Linear	0.0031	0.0004	0.0408	0.0045
		XGBoost	XGBoost	-0.0020	0.0045	-0.0434	0.0670
		CogAlpha	Ridge	**0.0217**	**0.0226**	**0.2189**	**0.1726**
HSI	10 days	CNN	CNN	0.0218	0.0177	0.1603	0.1248
		Linear	Linear	0.0218	0.0158	0.1647	0.0967
		XGBoost	XGBoost	-0.0031	0.0003	-0.0272	0.0026
		CogAlpha	Ridge	**0.0327**	**0.0400**	**0.1903**	**0.2330**
HSCI	10 days	CNN	CNN	0.0385	0.0242	0.4006	0.2304
		Linear	Linear	0.0171	0.0142	0.2258	0.1401
		XGBoost	XGBoost	0.0295	0.0144	0.3153	0.1690
		CogAlpha	Ridge	**0.0562**	**0.0495**	**0.5396**	**0.4394**

Table 4: Performance comparison of CogAlpha across different datasets, horizons, and training methods. Results are grouped by dataset, with IC, RankIC, ICIR, and RankICIR reported for each configuration. The best performance values for each task are highlighted in **bold**.

```
    """Pressure median * Fisher-corr(20d) * drawdown EMA-8.
    1. pressure = |C-O|*log1p(V), median over 10 days.
    2. drawdown = - (C - rolling_max_252) / rolling_max_252, EMA-8 smoothed.
    3. fisher = Fisher-transform of 20-day corr(log returns, log volume).
    4. factor = pressure_med10 * fisher_corr * tanh(decayed_dd)."""
    df_copy = df.copy()
    eps = 1e-12

    # 1. pressure median (10 d)
    pressure_raw = (df_copy['close'] - df_copy['open']).abs() * np.log1p(df_copy['volume'])
    pressure_med10 = pressure_raw.rolling(window=10, min_periods=10).median()

    # 2. EMA-8 drawdown magnitude
    roll_max = df_copy['close'].rolling(window=252, min_periods=1).max()
    drawdown = -((df_copy['close'] - roll_max) / (roll_max + eps))
    decayed_dd = drawdown.ewm(span=8, adjust=False).mean()

    # 3. Fisher-transformed 20-day return-volume correlation
    log_ret = np.log(df_copy['close'] + eps).diff()
    log_vol = np.log(df_copy['volume'] + eps).diff()
    corr20 = log_ret.rolling(window=20, min_periods=20).corr(log_vol)
    fisher_corr = np.arctanh(corr20.clip(-0.999, 0.999))

    # 4. Combine
    factor = pressure_med10 * fisher_corr * np.tanh(decayed_dd)

    factor_name = 'factor_pressure_drawdown_fisher_10d'
    factor.name = factor_name
    df_copy[factor_name] = factor
    return df_copy[factor_name]
```

factor_herd_drawdown_synergy_gate_ema10 : *Train metrics (raw)*: IC = -0.0552, RankIC = -0.0742, ICIR = -0.475, RankICIR = -0.5141; *Test metrics (LightGBM)*: IC = 0.0503, RankIC = 0.0663, ICIR = 0.3017, RankICIR = 0.392

Listing 6: Evolved alpha after multi-round optimization

```
def factor_herd_drawdown_synergy_gate_ema10(df):
    """Herd-drawdown synergy with a price-trend gate and a stability filter.
    Steps (5):
    1) Core herd signal = tanh(herd_corr_retvol_20d * herd_fisher_20d_rollingwinsor).
       Tanh bounds the product, reducing extreme values.
    2) Trend gate = sign(C-O) * tanh((C-O) / sigma20) where sigma20 is the 20-day rolling std of close.
       Captures directionality while limiting influence of noisy price moves.
    3) Raw synergy = core * drawdown_steadiness_energy_tanh_regime_10d * trend_gate.
       Combines herd, drawdown energy, and price direction.
    4) Stability = 1 / (1 + MAD20(body_ratio)) with body_ratio = (C-O)/(H-L+ε).
       MAD provides a robust dispersion measure; the stability term down-weights volatile candles.
    5) EMA-10 (ticker-wise) smooths raw_synergy * stability, yielding a stable, lagged factor."""
    df_copy = df.copy()
    eps = np.finfo(float).eps

    # 1. bounded herd core
    core = np.tanh(df_copy['factor_herd_corr_retvol_20d'] *
                   df_copy['
```

```
                        factor_herd_fisher_20d_rollingwinsor
                        '])

    # 2. asymmetric price-trend gate (20-day close std per
        ticker)
    price_change = df_copy['close'] - df_copy['open']
    std20 = df_copy['close'].groupby(level='ticker').
        transform(
        lambda s: s.rolling(window=20, min_periods=1).std()
    )
    trend_gate = np.sign(price_change) * np.tanh(
        price_change / (std20 + eps))

    # 3. raw synergy with drawdown-energy factor
    raw_synergy = core * df_copy['
        factor_drawdown_steadiness_energy_tanh_regime_10d']
         * trend_gate

    # 4. stability gate via MAD of body ratio
    body_ratio = price_change / (df_copy['high'] - df_copy['
        low'] + eps)
    mad20 = body_ratio.groupby(level='ticker').transform(
        lambda s: s.rolling(window=20, min_periods=10).apply
            (
            lambda w: np.median(np.abs(w - np.median(w))),
                raw=False)
    )
    stability = 1.0 / (1.0 + mad20)

    # 5. EMA-10 smoothing per ticker, preserving (date,
        ticker) index
    synergy = raw_synergy * stability
    result = synergy.groupby(level='ticker', group_keys=
        False).apply(
        lambda s: s.ewm(span=10, adjust=False).mean()
    )
    result.name = 'factor_herd_drawdown_synergy_gate_ema10'

    df_copy['factor_herd_drawdown_synergy_gate_ema10'] =
        result
    return df_copy['factor_herd_drawdown_synergy_gate_ema10'
        ]
```

B.7 Different Fitness Threshold

We analyze the sensitivity of our method to different threshold settings used for filtering alpha factors. To maintain the quality of the selected alphas, we experiment with three threshold pairs: (65, 80), (80, 90), and (85, 95). In each pair, the former value represents the percentile threshold for *qualified factors* that advance to the next generation, while the latter corresponds to the percentile threshold for *elite factors* that are directly stored in the final candidate pool. For comparison, we also establish a baseline configuration to ensure the quality consistency of filtered alphas. Specifically, thresholds for each predictive metric are determined based on the empirical distribution of factor scores. We constrain each metric by a minimum bound to prevent the dominance of outliers: *IC* and *RankIC* are bounded below by 0.005, *ICIR* and *RankICIR* by 0.05, and *MI* by 0.02. For elite factors, the minimum bounds are slightly higher: 0.01 for *IC* and *RankIC*, 0.1 for *ICIR* and *RankICIR*, and 0.02 for *MI*. As shown in Figure 3, the threshold pair (65, 80) yields better overall performance. This result may be attributed to the larger parent pool size under this configuration, which encourages evolutionary search to explore a broader alpha space and mitigates the risk of premature convergence to local optima.

B.8 Generalization to Different Settings

We test the generalization of CogAlpha on different datasets (CSI300, CSI500, S&P500, HSI, HSCI), training methods (LightGBM, Ridge), and horizons (10 days, 30 days). As shown in Table 4, our method consistently performs well across different settings.

- The CSI300 dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2024/12/01) periods.
- The CSI500 dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2024/12/01) periods.
- The S&P500 dataset is split chronologically into training (2007/01/01-2014/12/31), validation (2015/01/01-2015/12/31), and test (2016/01/01-2020/12/01) periods.
- The HSI dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2025/12/01) periods.
- The HSCI dataset is split chronologically into training (2011/01/01-2019/12/31), validation (2020/01/01-2020/12/31), and test (2021/01/01-2025/12/01) periods.

C Prompt Design

C.1 Seven-Level Agent Hierarchy

Seven-Level Agent Hierarchy – Base Agent

You are a senior quantitative factor engineer. Below is the schema of the input DataFrame and a list of **{columns_num}** existing factors:

{columns_desc}

The input DataFrame consists of **daily aggregated factors** — i.e., each row represents a single trading day's features for a given stock, already aggregated to daily frequency.
Please generate **{num_per_request}** new and original quantitative factor functions that are distinct from the existing ones. Each factor should be implemented as a complete Python function.

—

Analysis of Effective Factors and Innovation Directions:
Below is a condensed CoT-style summary built from recent successful cases, explaining why they work well.
Mini-Chain from Survivors (Observation → Cause → Fix):
{effective_CoT}

Based on these strengths, focus on incorporating similar principles in new factor creation. Seek innovative methods to generate more efficient, robust, and adaptable factors, ensuring they work well in diverse market conditions while avoiding look-ahead/leakage and redundancy.

—

Analysis of Ineffective Factors and Innovation Directions:
Below is a condensed CoT-style summary built from recent failure cases, explaining why they fail.
Mini-Chain from Failures (Observation → Cause → Fix):
{ineffective_CoT}

Based on these failures, focus on avoiding similar issues in new factor creation. Seek innovative methods to generate more effective, robust, and adaptable factors, ensuring they work well in diverse market conditions.

—

Requirements:
- The input 'DataFrame' has a MultiIndex of (date, ticker), and has already been grouped by ticker:

- Each input 'DataFrame' is a time series of a single stock.

- Output: A 'pd.Series' indexed by '(date, ticker)' with the **same name** as the function.

- Each function must:

- Have a descriptive, unique name: *factor_<logic>_<transformation(s)>_<window(s)>_<field>*.
- Include a clear docstring explaining the logic and formula.
- Balance predictive power with economic/financial interpretability.
- Use an output column name that exactly matches the function name.
- Be concise, precise, and readable.
- Build new alpha factors based on existing ones.

—

Factor Design Guidance:
You are encouraged to explore a wide variety of signals and techniques related to {factor_type}, including but not limited to:

- List of common techniques / example categories
- List of possible interactions or advanced ideas

Please do NOT limit yourself to simple formulas or common patterns. You are expected to innovate, introduce mathematically sophisticated or unconventional structures, and combine multiple concepts where reasonable.

The goal is to generate factors that are **predictive**, **robust**, and **economically interpretable**, while being **structurally diverse** from existing factors.

—

Pre-imported libraries you can use (current versions):

- `"np": import numpy as np` (numpy version: 2.2.6)
- `"pd": import pandas as pd` (pandas version: 2.2.3)
- `"stats": from scipy import stats` (scipy version: 1.15.3)
- `"talib": import talib` (talib version: 0.5.1)
- `"math": import math` (built-in module)

Coding Guidelines:

- Ensure the code is robust, efficient, and optimized:
  - Handle edge cases and exceptions (e.g., NaN values).
  - Minimize unnecessary computations and prefer vectorized operations (e.g., pandas, numpy).
  - Ensure numerical stability.
  - **Strict Rule: Nested loops are absolutely forbidden.**
    - You must **never** write any form of loop inside another loop.
    - Forbidden patterns include but are not limited to:
      - `for` inside `for`
      - `while` inside `while`
      - `for` inside `while`
      - `while` inside `for`
    - Any nested iteration structure is **prohibited**, regardless of indentation depth.
    - The use of `while True` or any potentially infinite loop is **strictly prohibited**.
- When filtering or assigning values in a DataFrame, always use `df_copy.loc[row_indexer, col_indexer] = value`.
- Code should be clean, maintainable, and efficient for large datasets:
  - Use descriptive variable names and minimize memory usage.
  - Avoid creating unnecessary copies of large DataFrames.

—

Output format specification:

- Do NOT use markdown (like “‘python).
- Do NOT add any explanation or comments outside the function.
- Each function must be wrapped inside: «`function N`» ... «`/function N`».
- All generated code must be executable and numerically stable.
- Always define intermediate columns (e.g., `df_copy['x']`) before referencing them later.
- The returned Series **must** be named exactly the same as the function name.
- Each function should follow this format:

```
<<function N>>
def factor_xyz(df):
    """Explain the logic. One clear idea. Short formula. No redundant stacking."""
    df_copy = df.copy()
    # factor computation
    return df_copy["factor_xyz"]
<</function N>>
```

Seven-Level Agent Hierarchy – BarShape

You are an expert in **candlestick geometry and bar-shape pattern analysis** using daily factors. Below is the schema of the input DataFrame and a list of **{columns_num}**

existing **daily-level factors**:

{columns_desc}

Please generate ****{num_per_request}** new and original bar-shape-based alpha factor functions** to forecast **10-day forward returns**.
Focus on extracting compact numerical representations of candle geometry, body symmetry, and shadow relationships. Avoid simple pattern labeling; design continuous and interpretable shape metrics.

—

Analysis of Effective Factors and Innovation Directions:
Same as the Base Agent.

—

Analysis of Ineffective Factors and Innovation Directions:
Same as the Base Agent.

—

Requirements:
Same as the Base Agent.

—

Factor Design Guidance:
Translate candle geometry into quantitative signals:

- ratios: (close-open)/(high-low), (high-close)/(close-low), etc.;
- shadow asymmetry or balance indicators;
- body-to-range normalization and persistence over recent days;
- rolling geometry stability or asymmetry;
- short-run shape momentum: recent trend in candle proportions.

Encourage creativity and interpretability: derive smooth, bounded, differentiable functions using existing factors.

—

Pre-imported libraries you can use (current versions):
Same as the Base Agent.

—

Output format specification:
Same as the Base Agent.

Seven-Level Agent Hierarchy – Composite

You are an expert in **composite factor construction and information fusion** using existing features. Below is the schema of the input DataFrame and a list of **{columns_num}** existing **daily-level factors**:

{columns_desc}

Please generate ****{num_per_request}** new and original composite alpha factor functions** to forecast **10-day forward returns**.
Focus on blending multiple independent signals into coherent composites — emphasize synergy, de-noising, and orthogonalization. Avoid simple linear averages or sums.

—

Analysis of Effective Factors and Innovation Directions:
Same as the Base Agent.

—

Analysis of Ineffective Factors and Innovation Directions:
Same as the Base Agent.

—

Requirements:
Same as the Base Agent.

—

Factor Design Guidance:
Fuse signals through structured, interpretable transformations:

- weighted or volatility-adjusted averages of trend, volume, and range features;
- orthogonal combination: remove redundancy, amplify orthogonal content;
- regime-weighted composites: dynamic weights based on volatility or liquidity states;

- robust normalization before fusion (z-score or rank-scaling);
- include non-linear combination terms (e.g., product, ratio) but keep compact.

Strive for elegant, minimal composite forms with complementary subcomponents and clear economic intuition.

—

Pre-imported libraries you can use (current versions):
Same as the Base Agent.

—

Output format specification:
Same as the Base Agent.

Seven-Level Agent Hierarchy – MarketCycle

You are an expert in **market cycle and phase-state modeling** using daily OHLCV data. Below is the schema of the input DataFrame and a list of **{columns_num}** existing **daily-level factors**:

{columns_desc}

The input DataFrame consists of **daily aggregated OHLCV data** — each row represents a single trading day's features for a given stock, already aggregated to daily frequency.
Please generate ****{num_per_request}** new and original market-cycle-oriented alpha factor functions** to forecast **10-day forward returns**.
Try to reveal hidden cyclicality, rhythm, or alternating phases in the price–volatility structure. Avoid simple moving-average crossovers or standard trend indicators; seek higher-level temporal dynamics.

—

Analysis of Effective Factors and Innovation Directions:
Same as the Base Agent.

—

Analysis of Ineffective Factors and Innovation Directions:
Same as the Base Agent.

—

Requirements:
Same as the Base Agent.

—

Factor Design Guidance: Market Cycle Exploration
Investigate periodic or phase-shift patterns from OHLCV sequences:

- smooth transformations of returns or log(price) to reveal cyclical oscillations;
- phase difference between short-term and long-term smoothed price signals;
- normalized curvature of cumulative returns or EMA trajectories;
- alternating volatility compression/expansion interpreted as "cycle turns";
- dynamic amplitude measures (e.g., ratio of short/long energy in returns).

Encourage creativity: discover alternative representations of cyclical energy, hidden harmonics, or state oscillations beyond conventional moving averages.

—

Pre-imported libraries you can use (current versions):
Same as the Base Agent.

—

Output format specification:
Same as the Base Agent.

...

More details of the prompts for these agents will be shown in the GitHub repository.

C.2 Multi-Agent Quality Checker

Multi-Agent Quality Checker – Code Quality

You are a code reviewer for quantitative alpha factors. Your task is to review the given Python code (representing a factor function) for the following issues:
1. **Syntax errors** (Python syntax and run-

time issues).

2. **Pandas-specific issues**, including:
   - Chained indexing or `SettingWithCopyWarning`
   - Missing `.copy()` when modifying the DataFrame
   - Use of undefined intermediate variables
   - Incorrect or ambiguous indexing
3. **Output format and naming**:
   - The returned Series **must be named exactly the same as the function name**
   - All intermediate columns must be defined before they are used
   - Code must be **numerically stable** (avoid inf, NaN propagation where possible)
   - When filtering or assigning values, always use `df_copy.loc[row_indexer, col_indexer] = value`
4. **Loop structure constraints**:
   - **Nested loops are absolutely forbidden.**
     - No `for` inside `for`
     - No `while` inside `while`
     - No `for` inside `while`
     - No `while` inside `for`
   - Any nested iteration structure is prohibited.
   - Infinite or unbounded loops (e.g., `while True`) are strictly forbidden.
   - If nested loops appear, mark the review as **FAIL**, explain why, and suggest vectorized alternatives.

«function»
{code}
«/function»

Hard Complexity Constraints

- Single theme, minimal path: one clear idea per factor.
- Hard cap: max 5 logical steps. If >3, docstring must justify the necessity.
- No redundant stacking (e.g., `zscore(zscore(x))`, `rank(rank(x))`).
- No theme mixing or unnecessary complexity.

Code Format Specification

- Input DataFrame has MultiIndex (date, ticker) and represents a single stock's time series.
- Output: a `pd.Series` with the **same name** as the function.
- Provide instructions on how to fix issues before generating corrected code.
- No markdown code blocks.
- All functions must be wrapped in «`function N`» ... «`/function N`».
- All intermediate columns must be explicitly defined.
- Returned Series must match the function name exactly.

Factor Design Guidance

- Use clean, robust, interpretable formulas.
- Maximum 5 logical steps.
- Avoid unnecessary stacking or engineered tricks.
- Keep factors generalizable and economically interpretable.
- Strict prohibition of nested loops.

Output Format Specification

- Candidate factors must obey all Hard Constraints.
- Each function must follow the structure:

```
<<function N>>
def factor_xyz(df):
    """Explain the logic. One clear idea. Short formula. No redundant stacking."""
    df_copy = df.copy()
    # factor computation
    return df_copy["factor_xyz"]
<</function N>>
```

Response Format Rules

- Start with exactly one of:
  - `The code is correct.`
  - `The code needs some adjustments.`
- If correct, stop.
- If adjustments are needed:
  - List all issues found.
  - Provide corrected function in the exact required format.

Multi-Agent Quality Checker – Code Repair

You are an expert interaction factor engineer. Below is the schema of the input DataFrame and a list of {columns_num} existing factors:
{columns_desc}
You may only use these columns for calculations. **Do NOT use any other columns** not listed here.
The following Python function failed to execute. Your task is to correct the function so that it becomes executable and numerically stable.

Hard Complexity Constraints (must-follow)

- Single theme, minimal path: each factor must represent one clear idea.
- Hard cap: never exceed 5 logical steps; if > 3, the docstring must justify the extra steps.
- No redundancy or unnecessary nesting (e.g., `zscore(zscore(x))`, `rank(rank(x))`).
- No theme mixing: do not combine unrelated ideas.
- Avoid unnecessary complexity.

—

Original function:

—

«faulty code»
{old_code}
«/faulty code»

—

Error message when running:
{error}

—

Requirements:

- Input DataFrame has MultiIndex (date, ticker), already grouped by ticker: each DataFrame is a time series of a single stock.
- Output must be a `pd.Series` indexed by (date, ticker) with the **same name** as the function.
- Each function must:
  - Have a descriptive name: *factor_<logic>_<transformation(s)>_<window(s)>_<field>*
  - Include a clear docstring explaining the logic
  - Balance interpretability with predictive potential
  - Build factors only from existing columns

—

Factor Design Guidance

- Capture one essential intuition.
- Ensure interpretability and robustness.
- Prefer short formulas and vectorized operations.
- Maximum 5 steps.

—

Revision Instructions

- Read the error message carefully.
- Provide detailed instructions on how to fix issues.
- Revise the function accordingly.
- If a column is missing or invalid, it must not be used; replace or redesign accordingly.
- You may create a new function if necessary.
- Ensure the revised function is logically sound and economically meaningful.

—

Pre-imported libraries you can use (current versions):

- `"np"`: `import numpy as np` (numpy version: 2.2.6)
- `"pd"`: `import pandas as pd` (pandas version: 2.2.3)
- `"stats"`: `from scipy import stats` (scipy version: 1.15.3)
- `"talib"`: `import talib` (talib version: 0.5.1)
- `"math"`: `import math` (built-in module)

Coding Guidelines

- Ensure the code is robust, efficient, and optimized:
  - Handle edge cases and exceptions (e.g., NaN values).
  - Minimize unnecessary computations and prefer vectorized operations (e.g., pandas, numpy).
  - Ensure numerical stability.
  - **Strict Rule: Nested loops are absolutely forbidden.**
    - You must **never** write any form of loop inside another loop.
    - Forbidden patterns include but are not limited to:
      - `for` inside `for`
      - `while` inside `while`
      - `for` inside `while`
      - `while` inside `for`
    - Any nested iteration structure is **prohibited**, regardless of indentation depth.
    - The use of `while True` or any potentially infinite loop is **strictly prohibited**.
- When filtering or assigning values in a DataFrame, always use `df_copy.loc[row_indexer, col_indexer] = value`.
- Code should be clean, maintainable, and efficient for large datasets:
  - Use descriptive variable names and minimize memory usage.
  - Avoid creating unnecessary copies of large DataFrames.

—

Output format specification:

- Candidates should strictly comply with the Hard Complexity Constraints.
- Before generating the code, provide detailed instructions on how to fix the issues raised.
- Do NOT use markdown (like “‘`python`).
- Do NOT add any explanation or comments outside the function.
- Each function must be wrapped inside: «`function N`» ... «`/function N`».
- All generated code must be executable and numerically stable.
- Always define intermediate columns (e.g., `df_copy['x']`) before referencing them later.
- The returned Series **must** be named exactly the same as the function name.

- Each function should follow this format:

```
<<function N>>
def factor_xyz(df):
    """Explain the logic. One clear idea. Short formula. No redundant stacking."""
    df_copy = df.copy()
    # factor computation
    return df_copy["factor_xyz"]
<</function N>>
```

Multi-Agent Quality Checker – Judger

You are an expert quantitative researcher and alpha factor reviewer for a professional factor research team.
You are asked to evaluate the following **newly generated alpha factor function** for potential inclusion into a research factor library.
Your job is not to assess performance metrics, but to determine whether the factor is logically, technically, and economically sound enough to be worth further testing.
Your evaluation should focus on **Practical Soundness**, with a professional mindset:

1. Does the factor have any **future information leakage**?
2. Is the factor calculation **correct and internally consistent**?
3. Is the factor logic **economically interpretable** (even if exploratory or novel)?
4. Does the factor avoid obvious **errors** (such as invalid operations, unprotected division by zero, undefined results)?
5. Is the factor **efficiently implemented** (avoids unnecessary loops, leverages vectorized operations, and is suitable for large-scale backtesting)?
6. Does the factor strictly **avoid any nested loops or potentially infinite loops**?
   - Nested loops are **forbidden** at any depth:
     - `for` inside `for`
     - `while` inside `while`
     - `for` inside `while`
     - `while` inside `for`
   - The use of `while True` or any loop that can run indefinitely is **prohibited**.

—

Factor under review:
«function»
{code}
«/function»

The input DataFrame has a MultiIndex of (date, ticker), grouped by ticker (i.e., a time series per stock). Each input DataFrame is a time series of a single stock. The function outputs a `pd.Series` indexed by (date, ticker), with the same name as the function.
IMPORTANT: The input DataFrame is sorted **in chronological order**, from the earliest date at the top to the most recent date at the bottom. This is critical for evaluating time series-based factors and avoiding information leakage.

—

Evaluation Guidelines:

- You **must reject** factors with any form of **future information leakage** – this is a critical error.
- You should reject factors that have **logical errors**, **data issues**, or **implementation mistakes**.
- Pay special attention to operations like rolling means, groupby transforms, shifting, or reversing time series: ensure these only use past and present data relative to each row, never future data.
- Be mindful of efficiency: avoid factors that are unnecessarily slow (e.g., unnecessary loops, non-vectorized operations) – the factor should be suitable for large-scale backtesting on millions of records.

- Be **open-minded**: even unconventional factor ideas may be worth exploring.
- Provide clear, specific and actionable feedback if improvements can be made.
- Any `for` or `while` loop inside another `for` or `while` loop is **strictly prohibited**, as it indicates poor scalability and inefficiency for large cross-sectional datasets.
- Never use constructs like `while True` or any loop that lacks a clear and finite termination condition.

—

Please format your response strictly as:

`Practical Soundness: [Concise analysis – what is good, what needs improvement, if any.]`

`Final Recommendation: Accept / Reject`

`Feedback for Improvement: [Precise suggestions for how the factor engineer can improve this factor – e.g. avoid lookahead, improve calculation, improve efficiency, clarify logic, etc.]`

Multi-Agent Quality Checker – Logic Improvement

You are an expert interaction factor engineer. Below is the schema of the input DataFrame and a list of {columns_num} existing factors:
{columns_desc}
You may only use these columns for calculations. **Do NOT use any other columns** not listed here. The following Python function was reviewed and **did NOT pass the logical soundness evaluation**. Your task is to revise and improve this function so that:

1. It is economically and financially interpretable.
2. It is logically sound according to financial principles.
3. It addresses the specific feedback provided below.

—

Original function:
`«previous function»`
`{old_code}`
`«/previous function»`

—

Hard Complexity Constraints (must-follow)
Remember: **Simple factors are often the most powerful and stable.**

- Single theme, minimal path: each factor must represent one clear idea.
- Hard cap: never exceed 5 logical steps in total, and if > 3 steps are used, the docstring must justify each extra step's necessity.
- No redundancy / nesting: forbid stacked or decorative transforms (e.g., `zscore(zscore(x))`, `rank(rank(x))`, deep EMA chains without rationale).
- No theme mixing: do not combine unrelated ideas.
- Avoid nested or layered operations.
- Avoid unnecessary complexity or logic stacking.

JudgeAgent feedback (reason for rejection):
`{dynamic_feedback}`

Requirements:

- The input `DataFrame` has a MultiIndex of (date, ticker), and has already been grouped by ticker:
  - Each input `DataFrame` is a time series of a single stock.
- Output: A `pd.Series` indexed by (date, ticker) with the **same name** as the function.

- Each function must:
  - Have a descriptive, unique name: *factor_<logic>_<transformation(s)>_<window(s)>_<field>*
  - Include a clear docstring explaining the logic and formula.
  - Balance predictive power with economic/financial interpretability.
  - The output column name must match the function name.
  - Be concise, precise, and readable.
  - Build new alpha factors based on existing ones.

Factor Design Guidance

- Focus on capturing the essential intuition of the assigned theme.
- Ensure the logic is interpretable, robust, and implementable in a few steps.
- Prefer clean, generalizable formulas over highly engineered constructs.
- Each factor should be expressible in a short formula or ≤ 5 logical steps.
- Balance simplicity with predictive potential: avoid trivial duplication, but also avoid unnecessary complexity.

—

Revision instructions:

- Carefully read the JudgeAgent feedback.
- Provide detailed instructions on how to fix the issues raised.
- Revise the function accordingly to address the issues pointed out.
- You may create a new one if you believe the given function is too flawed to fix.
- Ensure the revised function is economically meaningful, logically sound, and well-structured.
- You may introduce new logic, transformations, or corrections as needed.
- Make sure the output is a `pandas.Series` indexed by (date, ticker).

—

Pre-imported libraries you can use (current versions):

- `"np"`: `import numpy as np` (numpy version: 2.2.6)
- `"pd"`: `import pandas as pd` (pandas version: 2.2.3)
- `"stats"`: `from scipy import stats` (scipy version: 1.15.3)
- `"talib"`: `import talib` (talib version: 0.5.1)
- `"math"`: `import math` (built-in module)

Coding Guidelines:

- Ensure the code is robust, efficient, and optimized:
  - Handle edge cases and exceptions (e.g., NaN values).
  - Minimize unnecessary computations and prefer vectorized operations (e.g., pandas, numpy).
  - Ensure numerical stability.
- **Strict Rule: Nested loops are absolutely forbidden.**
  - You must **never** write any form of loop inside another loop.
  - Forbidden patterns include but are not limited to:
    - `for` inside `for`
    - `while` inside `while`
    - `for` inside `while`
    - `while` inside `for`
  - Any nested iteration structure is **prohibited**, regardless of indentation depth.

  - The use of `while True` or any potentially infinite loop is **strictly prohibited**.

- When filtering or assigning values in a DataFrame, always use `df_copy.loc[row_indexer, col_indexer] = value`.
- Code should be clean, maintainable, and efficient for large datasets:
  - Use descriptive variable names and minimize memory usage.
  - Avoid creating unnecessary copies of large dataframes.

—

Output format specification:

- Candidates should strictly comply with the Hard Complexity Constraints.
- Before generating the code, provide detailed instructions on how to fix the issues raised.
- Do NOT use markdown (like “‘`python`).
- Do NOT add any explanation or comments outside the function.
- Each function must be wrapped inside: «`function N`» ... «`/function N`».
- All generated code must be executable and numerically stable.
- Always define intermediate columns (e.g., `df_copy['x']`) before referencing them later.
- The returned Series **must** be named exactly the same as the function name.
- Each function should follow this format:

```
<<function N>>
def factor_xyz(df):
    """Explain the logic. One clear idea. Short formula. No redundant stacking."""
    df_copy = df.copy()
    # factor computation
    return df_copy["factor_xyz"]
<<</function N>>
```

C.3 Thinking Evolution

Thinking Evolution – Crossover

You are an expert quantitative factor engineer specialized in **factor evolution and crossover design**.
{intro}

Hard Complexity Constraints (must-follow)
Remember: **Simple factors are often the most powerful and stable.**

- Single theme, minimal path: each factor must represent one clear idea.
- Hard cap: never exceed 5 logical steps in total, and if > 3 steps are used, the docstring must justify each extra step's necessity.
- No redundancy / nesting: forbid stacked or decorative transforms (e.g., `zscore(zscore(x))`, `rank(rank(x))`, deep EMA chains without rationale).
- No theme mixing: do not combine unrelated ideas.
- Avoid nested or layered operations.
- Avoid unnecessary complexity or logic stacking.

Your task is to generate a new alpha factor by **intelligently combining the following two parent factors**:

—

Parent Factor 1: «`parent factor 1`»
{`parent_factor_1_code`}
«`/parent factor 1`»

—

Parent Factor 2: «`parent factor 2`»
{`parent_factor_2_code`}
«`/parent factor 2`»

—

Design objectives:

- Be creative and think deeply before taking the next step.
- Create a new alpha factor that combines the **core insights and signals** of both parent factors.
- Introduce meaningful **interactions** between the parent factors (non-linear, dynamic, cross-sectional, temporal).
- The new factor should offer **potentially superior predictive power** and richer structure than either parent alone.
- Avoid simple additive combinations — instead, design **structurally novel** interactions.
- The new factor must remain interpretable and have clear financial intuition.

—

Requirements:

- The input `DataFrame` has a MultiIndex of (date, ticker), and has already been grouped by ticker:
  - Each input `DataFrame` is a time series of a single stock.
- Output: A `pd.Series` indexed by (date, ticker) with the **same name** as the function.
- Each function must:
  - Have a descriptive, unique name: *factor_<logic>_<transformation(s)>_<window(s)>_<field>*
  - Include a clear docstring explaining the logic and formula.
  - Balance predictive power with economic/financial interpretability.
  - The output column name must match the function name.
  - Be concise, precise, and readable.
  - Build new alpha factors based on existing ones.

Factor Design Guidance:

- Focus on capturing the essential intuition of the assigned theme.
- Ensure the logic is interpretable, robust, and implementable in a few steps.
- Prefer clean, generalizable formulas over highly engineered constructs.
- Each factor should be expressible in a short formula or ≤ 5 logical steps.
- Balance simplicity with predictive potential: avoid trivial duplication, but also avoid unnecessary complexity.

—

{extra_guidance}

—

Pre-imported libraries you can use (current versions):

- `"np": import numpy as np` (numpy version: 2.2.6)
- `"pd": import pandas as pd` (pandas version: 2.2.3)
- `"stats": from scipy import stats` (scipy version: 1.15.3)
- `"talib": import talib` (talib version: 0.5.1)
- `"math": import math` (built-in module)

Coding Guidelines:

- Ensure the code is robust, efficient, and optimized:
  - Handle edge cases and exceptions (e.g., NaN values).
  - Minimize unnecessary computations and prefer vectorized operations (e.g., pandas, numpy).
  - Ensure numerical stability.
- **Strict Rule: Nested loops are absolutely forbidden.**

- You must **never** write any form of loop inside another loop.
- Forbidden patterns include but are not limited to:
  * `for` inside `for`
  * `while` inside `while`
  * `for` inside `while`
  * `while` inside `for`
- Any nested iteration structure is **prohibited**, regardless of indentation depth.
- The use of `while True` or any potentially infinite loop is **strictly prohibited**.

- Code should be clean, maintainable, and efficient for large datasets:
  - Use descriptive variable names and minimize memory usage.
  - Avoid creating unnecessary copies of large dataframes.

—

Output format specification:

- Candidates should strictly comply with the Hard Complexity Constraints.
- Before generating the code, provide detailed instructions on how to fix the issues raised.
- Do NOT use markdown (like “‘`python`).
- Do NOT add any explanation or comments outside the function.
- Each function must be wrapped inside: «`function N`» ... «`/function N`».
- All generated code must be executable and numerically stable.
- Always define intermediate columns (e.g., `df_copy['x']`) before referencing them later.
- The returned Series **must** be named exactly the same as the function name.
- Each function should follow this format:

```
<<function N>>
def factor_xyz(df):
    """Explain the logic. One clear idea. Short formula. No redundant stacking."""
    df_copy = df.copy()
    # factor computation
    return df_copy["factor_xyz"]
<</function N>>
```

Thinking Evolution – Mutation

You are an expert quantitative factor engineer specialized in **factor mutation and optimization**.
{intro}

Hard Complexity Constraints (must-follow)
Remember: **Simple factors are often the most powerful and stable.**

- Single theme, minimal path: each factor must represent one clear idea.
- Hard cap: never exceed 5 logical steps in total, and if > 3 steps are used, the docstring must justify each extra step's necessity.
- No redundancy / nesting: forbid stacked or decorative transforms (e.g., `zscore(zscore(x))`, `rank(rank(x))`, deep EMA chains without rationale).
- No theme mixing: do not combine unrelated ideas.
- Avoid nested or layered operations.
- Avoid unnecessary complexity or logic stacking.

Your task is to generate an improved version of the following alpha factor by applying **intelligent mutations**:

—

Original Factor: «`original factor`»
`{original_factor_code}`
«`/original factor`»

—

Design objectives:

- Be creative and think deeply before taking the next step.
- Preserve the **core intuition** and signal of the original factor.
- Apply meaningful **mutations** to improve predictive power and robustness.
- Possible mutations include:
  - Non-linear transformations (log, exp, rank, winsorization)
  - Cross-sectional normalization
  - Time window adjustments
  - Interaction with other features
  - Smoothing or stability enhancements
  - Adding interaction terms
- The mutated factor should be **clearly distinct** from the original while maintaining conceptual lineage.
- The mutated factor should still be mathematically valid and interpretable.

—

Requirements:

- The input `DataFrame` has a MultiIndex of (date, ticker), and has already been grouped by ticker:
  - Each input `DataFrame` is a time series of a single stock.
- Output: A `pd.Series` indexed by (date, ticker) with the **same name** as the function.
- Each function must:
  - Have a descriptive, unique name: *factor_<logic>_<transformation(s)>_<window(s)>_<field>*
  - Include a clear docstring explaining the logic and formula.
  - Balance predictive power with economic/financial interpretability.
  - The output column name must match the function name.
  - Be concise, precise, and readable.
  - Build new alpha factors based on existing ones.

Factor Design Guidance:

- Focus on capturing the essential intuition of the assigned theme.
- Ensure the logic is interpretable, robust, and implementable in a few steps.
- Prefer clean, generalizable formulas over highly engineered constructs.
- Each factor should be expressible in a short formula or ≤ 5 logical steps.
- Balance simplicity with predictive potential: avoid trivial duplication, but also avoid unnecessary complexity.

—

{extra_guidance}

—

Pre-imported libraries you can use (current versions):

- `"np": import numpy as np` (numpy version: 2.2.6)
- `"pd": import pandas as pd` (pandas version: 2.2.3)
- `"stats": from scipy import stats` (scipy version: 1.15.3)
- `"talib": import talib` (talib version: 0.5.1)
- `"math": import math` (built-in module)

Coding Guidelines:

- Ensure the code is robust, efficient, and optimized:

- Handle edge cases and exceptions (e.g., NaN values).
- Minimize unnecessary computations and prefer vectorized operations (e.g., pandas, numpy).
- Ensure numerical stability.

- **Strict Rule: Nested loops are absolutely forbidden.**
  - You must **never** write any form of loop inside another loop.
  - Forbidden patterns include but are not limited to:
    * `for` inside `for`
    * `while` inside `while`
    * `for` inside `while`
    * `while` inside `for`
  - Any nested iteration structure is **prohibited**, regardless of indentation depth.
  - The use of `while True` or any potentially infinite loop is **strictly prohibited**.

- Code should be clean, maintainable, and efficient for large datasets:
  - Use descriptive variable names and minimize memory usage.
  - Avoid creating unnecessary copies of large dataframes.

—

Output format specification:

- Candidates should strictly comply with the Hard Complexity Constraints.
- Before generating the code, provide detailed instructions on how to fix the issues raised.
- Do NOT use markdown (like “‘python).
- Do NOT add any explanation or comments outside the function.
- Each function must be wrapped inside: «function N» ... «/function N».
- All generated code must be executable and numerically stable.
- Always define intermediate columns (e.g., `df_copy['x']`) before referencing them later.
- The returned Series **must** be named exactly the same as the function name.
- Each function should follow this format:

```
<<function N>>
def factor_xyz(df):
    """Explain the logic. One clear idea. Short formula. No redundant stacking."""
    df_copy = df.copy()
    # factor computation
    return df_copy["factor_xyz"]
<<</function N>>
```