ALPHAQT-BENCH: Diagnosing the Gap between Financial Code Generation and Quantitative Reasoning in LLMs

Sichun Luo[2], Yi Huang[1*], Shichang Meng[3], Fengyuan Liu[2], Mukai Li[2], Qinghua Yao[2], Zefa Hu[1], Junlan Feng[1], Qi Liu[2*]
[1]JIUTIAN Research, China Mobile [2]The University of Hong Kong
[3]City University of Hong Kong
sichunluo2@gmail.com

Abstract

Large Language Models (LLMs) are increasingly applied to alpha mining in quantitative finance, marking a shift from generating simple symbolic formulas to producing executable, code-based strategies. While code generation offers greater expressiveness, it introduces critical risks absent in symbolic approaches, including temporal causality violations (look-ahead bias) and stateful logic bugs. Existing benchmarks largely rely on outcome-driven metrics (e.g., backtesting profitability), which often conflate market stochasticity or unintended information leakage with genuine reasoning competence. We introduce **ALPHAQT-BENCH**, a diagnostic benchmark for instruction-grounded financial code generation under strict semantic and temporal constraints. Unlike general-purpose coding benchmarks, ALPHAQT-BENCH adopts a multi-layer evaluation protocol that assesses: (1) executability, (2) causality safety via a dynamic truncation test, (3) functional accuracy, and (4) structural compliance through vectorization analysis. Experiments across 12 representative LLMs reveal a substantial gap between surface-level success (e.g., executability) and verified quantitative correctness, as many models fail under causal, structural, or functional constraints. By shifting evaluation from *profitability* to *process reliability*, ALPHAQT-BENCH provides a principled safety audit for emerging LLM-based quantitative systems.

1 Introduction

Alpha mining lies at the core of quantitative finance, where the objective is to discover trading signals that generate excess returns (Fama and French, 1993; Zhang et al., 2025). Traditionally, this process relied on closed-form symbolic factors composed of predefined operators, offering limited expressiveness but strong implicit safety guarantees (Yu et al., 2023; Shi et al., 2025a). Recent advances in Large Language Models (LLMs) are reshaping this paradigm by enabling automated *code-based alpha generation*, in which alpha factors are implemented as executable programs that support more expressive operations, such as conditional logic and stateful transformations (Liu et al., 2025). While this transition substantially expands the space of representable strategies, it also introduces new challenges in verifying whether generated code reflects correct quantitative reasoning.

Recently, LLMs have demonstrated strong capabilities in general-purpose code generation (Li et al., 2022; Jiang et al., 2024), achieving superior performance on benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). However, success in generating runnable code does not necessarily imply correct quantitative reasoning. In the context of alpha mining, correctness extends beyond syntactic validity or numerical plausibility, and crucially depends on adherence to domain-specific principles that govern financial data and trading logic. Quantitative finance imposes strict semantic constraints that are largely invisible to general-purpose code evaluation. In particular, *Temporal Causality* prohibits any use of future information when computing trading signals, as even subtle look-ahead bias can invalidate empirical findings. In addition, practical evaluation of quantitative code often requires *Stateful Vectorization*, which enforces that state-dependent computations (e.g., conditional rolling statistics or exponential smoothing) be implemented using efficient, loop-free array operations rather than explicit iterative constructs that obscure data dependencies. Violations of either constraint can produce implementations that appear correct under execution-based metrics, yet fail to meet the quantitative or procedural standards required for reliable alpha construction.

Despite the rapid development of LLM-based

*Corresponding author

Findings of the Association for Computational Linguistics: ACL 2026, pages 2871–2886

alpha mining systems, existing evaluation practices remain largely outcome-driven (Ding et al., 2025; Luo et al., 2026). Much prior work assesses model performance using downstream profitability metrics, such as cumulative returns or information ratio (Tang et al., 2025; Shi et al., 2025b). We argue that such evaluation is misaligned with the goal of measuring reasoning competence. **Profitability can conflate correctness with stochastic effects**: an implementation that inadvertently violates causal constraints (e.g., by peeking at future data) may achieve inflated backtest performance, while a logically sound factor can underperform due to market noise. As a result, outcome-based metrics provide limited diagnostic insight, making it difficult to determine whether failures stem from incorrect implementation, semantic misunderstanding, or violations of fundamental safety constraints.

To fill this gap, we introduce **AlphaQT-Bench**, a diagnostic benchmark designed to shift evaluation from profitability to process reliability. Rather than asking whether a model can generate profit, AlphaQT-Bench assesses whether it can faithfully translate quantitative intent into causally valid and structurally compliant code. Tasks are instruction-grounded and hierarchically organized into three levels of difficulty, ranging from basic factor construction to complex algorithmic reasoning. Beyond dataset construction, we propose a *multi-layer evaluation protocol* that decomposes performance into interpretable dimensions. Specifically, the protocol evaluates (i) *Executability*, (ii) *Causality Safety* via a dynamic truncation test, (iii) *Functional Accuracy* against expert-written Golden Code, and (iv) *Structural Compliance* through vectorization analysis. Experiments across 12 representative LLMs reveal a substantial gap between surface-level code generation success and verified quantitative reasoning, with many models failing under causal or structural constraints despite producing runnable code.

In a nutshell, our contributions are threefold:

- We introduce **AlphaQT-Bench**, a curated dataset of quantitative tasks that exposes the gap between execution success and quantitative correctness, including failures caused by look-ahead bias and stateful logic bugs.
- We propose a multi-layer evaluation protocol that moves beyond execution success, diagnosing failures across syntax, causality safety, functional correctness, and structural compliance.
- Through extensive experiments, we reveal a systematic gap between surface-level code generation and robust quantitative reasoning, underscoring the need for domain-specific evaluation and alignment in LLMs.

2 Related Work

2.1 Alpha Mining: From Symbolic Discovery to Code Generation

Automated alpha mining remains a central challenge in quantitative finance. While traditional methods such as Genetic Programming (GP) (Lin et al., 2019; Su et al., 2022; Ren et al., 2024) have been widely adopted, they often suffer from overfitting and limited interpretability. The advent of LLMs has catalyzed a shift from heuristic search toward more flexible, generative approaches.

Much prior work treats alpha mining as a symbolic regression task, where LLMs synthesize mathematical expressions composed of predefined operators (e.g., `Rank(Ts_ArgMax(Close, 10))`). Representative frameworks include AlphaAgent (Tang et al., 2025), which employs regularization to mitigate alpha decay, and Chain-of-Alpha (Cao et al., 2025), which utilizes a dual-chain mechanism for joint generation and selection. These methods benefit from *implicit safety*: confinement to stateless, predefined operators inherently prevents many forms of temporal leakage. However, this safety comes at the cost of *limited expressiveness*, making it difficult to represent the complex, regime-switching, or stateful logic required in modern quantitative research pipelines.

To address the expressiveness limitations of symbolic representations, recent work has begun to explore agentic, code-based paradigms for quantitative strategy development. Systems such as RD-Agent (Q) (Li et al., 2025) introduce agentic research-and-development frameworks in which LLMs iteratively propose hypotheses and implement executable code within an end-to-end pipeline that integrates backtesting and feedback, while CogAlpha (Liu et al., 2025) investigates multi-level cognitive architectures for sustained, code-level alpha discovery. However, this increased flexibility introduces a critical new class of risks, which we refer to as *silent semantic errors*. Unlike rigid symbolic formulas, executable Python code permits subtle temporal causality violations (e.g., improper indexing or look-ahead bias) and stateful logic bugs that are difficult to detect via simple execution or

backtesting. While agentic systems primarily emphasize optimization and performance feedback loops, ALPHAQT-BENCH focuses on auditing the *reliability* of the code generation process itself.

2.2 Evaluation Paradigms: From Profitability to Process Reliability

As LLMs transition from generating text to generating executable strategies, evaluation paradigms should evolve from verifying syntactic correctness to assessing domain-specific reasoning fidelity.

General-purpose code benchmarks, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), evaluate Python proficiency but lack domain-specific constraints. They typically assess functional correctness via unit tests that are agnostic to time. Therefore, a model may achieve state-of-the-art performance on HumanEval while producing code that violates temporal causality (e.g., through centered normalization or improper indexing) in financial contexts, rendering the implementation unreliable despite being syntactically valid.

Recent alpha mining benchmarks, such as AlphaEval (Ding et al., 2025) and AlphaBench (Luo et al., 2026), incorporate market data to evaluate generated factors. However, these frameworks primarily assess performance through downstream backtesting metrics. Such outcome-based evaluation is often misaligned with the goal of measuring reasoning correctness. However, implementations containing subtle look-ahead bias may yield inflated returns, while logically rigorous strategies can underperform due to market stochasticity (Bailey and López de Prado, 2014; De Prado, 2018). As a result, performance alone provides limited diagnostic insight into whether the generated code adheres to correct temporal and semantic reasoning.

There remains a lack of benchmarks that explicitly audit the process reliability of quantitative code generation. ALPHAQT-BENCH addresses this gap by introducing a diagnostic evaluation protocol that verifies *causal safety* (via dynamic truncation tests) and *structural compliance* (e.g., vectorization), decoupling the assessment of implementation correctness from noisy market outcomes.

3 ALPHAQT-BENCH: Task and Evaluation Framework

This section formalizes ALPHAQT-BENCH as a diagnostic benchmark for instruction-grounded code generation under strict semantic and temporal constraints. We define the task setting, articulate the core constraints that govern correctness, describe the hierarchical dataset construction, and introduce a multi-layer evaluation protocol designed to isolate distinct failure modes in quantitative reasoning.

3.1 Task Definition: Instruction-Grounded Code Generation

We study the problem of *instruction-grounded quantitative code generation*, where a model translates a natural language specification into an executable code satisfies domain-specific constraints.
Input. The input is a natural language instruction S describing a quantitative transformation over a univariate time series. Instructions range from explicit specifications (e.g., fixed window sizes) to underspecified requirements that force the model to resolve implicit financial assumptions.
Output. The model must generate a Python function f that maps an input data tensor $\mathcal{D} \in \mathbb{R}^{T\times N}$, which contains historical observations such as open, high, low, close, and volume (OHLCV), to a one-dimensional signal series:

$$\mathbf{y} = f(\mathcal{D}), \quad \mathbf{y} \in \mathbb{R}^{T}. \tag{1}$$

The generated function must return a series aligned with the input timeline, where each output value y_t represents the signal state at time step t.
Correctness Criterion. Correctness criterion in ALPHAQT-BENCH is defined not merely by execution success, but by strict adherence to *temporal causality* and *semantic alignment* with the expert-verified ground truth.

3.2 Semantic and Temporal Constraints

Unlike general-purpose programming, time-series code generation is governed by global constraints that cannot be reliably verified through a finite set of local unit tests alone.

Temporal Causality as a Global Semantic Invariant. The primary constraint is **temporal causality**: the output at time t must depend *only* on information available up to that time. Formally, a causally valid implementation must satisfy:

$$y_t = f(\mathcal{D}_{0:t}), \quad \forall t \in [1, T]. \tag{2}$$

Any dependence on future observations $\mathcal{D}_{t+k}$ ($k > 0$) constitutes a *look-ahead bias*. Such violations are particularly insidious in practical code generation settings, as programs may remain syntactically

valid and executable while silently incorporating future information, rendering the strategy invalid.

Vectorization as a Structural Constraint for Deployable Quantitative Code. ALPHAQT-BENCH enforces a strict vectorization constraint: solutions must be expressed using array operations (e.g., Pandas/NumPy) rather than explicit iterative control flow. We impose this as a production-grade deployability requirement: loop-based implementations are empirically 600×–2000× slower than vectorized equivalents (Table 3), rendering them infeasible in realistic backtesting pipelines spanning thousands of assets. Beyond efficiency, this constraint also serves as a diagnostic signal for stateful temporal reasoning deficits—the inability to map recursive or windowed logic to vectorized primitives (e.g., `.ewm()`, `.rolling().apply()`) often reflects gaps in domain-specific reasoning about index alignment and state propagation.

3.3 Dataset Construction and Task Taxonomy

To disentangle surface-level code recall from genuine semantic reasoning, ALPHAQT-BENCH employs a hierarchical task taxonomy paired with expert-verified golden standards. Representative examples from each level are shown in Table 1.

3.3.1 Hierarchical Task Levels

Tasks are organized into three levels of reasoning abstraction, designed to separate surface-level code generation from genuine quantitative reasoning.

• **Level 1: Primitive Grounding.** Evaluates the ability to map explicit instructions to standard numerical primitives (e.g., moving averages). These tasks primarily test surface-level lexical grounding and library API competence, and can often be solved via direct pattern recall.

• **Level 2: Compositional Semantics.** Requires composing multiple primitives into coherent computational pipelines. Success depends on correctly handling intermediate representations (e.g., log-returns), index alignment, and multi-step transformations, exposing errors in compositional temporal reasoning and data flow composition.

• **Level 3: Algorithmic Reasoning.** Requires translating abstract, often underspecified logic into fully vectorized implementations without reliance on memorized templates. These tasks stress-test stateful logic, recursive dependencies, and complex windowed operations, and are intentionally constructed to resist solution via surface pattern matching. This level serves as the primary discriminator for genuine algorithmic reasoning failures.

Level 1: Primitive Grounding
Calculate the 10-day simple moving average of the daily closing price `day_close`.

```python
def sma_10(df):
    return df['day_close'].rolling(10).mean()
```

Level 2: Compositional Semantics
Compute the 20-day rolling volatility defined as the standard deviation of daily log returns.

```python
def rolling_vol_20(df):
    ret = np.log(df['day_close'] /
                 df['day_close'].shift(1))
    return ret.rolling(20).std()
```

Level 3: Algorithmic Reasoning
For each day t, compute the slope of a linear regression of `day_close` *against over the past 20 days.*

```python
def rolling_lr_slope(df):
    # Vectorized implementation of OLS
    x = np.arange(20, dtype=float)
    x_mean = x.mean()
    denom = ((x - x_mean) ** 2).sum()

    def slope(y):
        y_mean = y.mean()
        return ((x - x_mean) *
                (y - y_mean)).sum() / denom

    return df['day_close'].rolling(
        20, min_periods=20
    ).apply(slope, raw=True)
```

Table 1: Example tasks from ALPHAQT-BENCH, illustrating the progression from API grounding to complex algorithmic reasoning under vectorization constraints.

3.3.2 Human–LLM Co-Design

Tasks are constructed through a human-in-the-loop human–LLM co-design process that balances semantic precision with linguistic diversity. Domain experts first specify the intended quantitative semantics and causal constraints of each task. LLMs are then used to generate diverse natural language formulations of the same underlying objective, reducing spurious correlations between surface phrasing and solution patterns. Crucially, every task is paired with a *golden code* implementation developed through human–LLM collaboration and rigorously validated by domain experts. All golden code implementations are manually audited to ensure functional correctness, strict temporal causality, and compliance with vectorization constraints.

Annotator Profiles. The Golden Code was developed and validated by two domain experts, each

possessing more than two years of quantitative research experience. Notably, both experts also have prior experience in NLP dataset annotation, ensuring a dual competency in both financial domain logic and NLP dataset annotation.

Annotation Pipeline. The construction process consisted of three stages: (1) Logic definition. Each factor's mathematical definition and temporal constraints were formally specified before any code was written; (2) Human–LLM collaborative implementation. Golden code was developed through iterative human–LLM collaboration, with domain experts guiding the generation process and refining implementations through execution-based testing; (3) Manual audit and validation. All golden code implementations were manually audited by domain experts to ensure functional correctness, strict temporal causality, and compliance with vectorization constraints, with any discrepancies resolved through discussion until full consensus was reached.

Inter-annotator Agreement. Since the final golden code is consensus-based rather than independently labeled, traditional code-level inter-annotator agreement is not directly applicable. To quantify annotation consistency, we designed a category classification task as a proxy measure. Specifically, we randomly sampled 10% of the dataset along with their category labels, shuffled the label assignments, and asked both annotators to independently judge the correctness of each label (binary classification). We then computed Cohen's Kappa to assess agreement beyond chance. The resulting Cohen's Kappa = 0.7781, indicating substantial agreement (Landis and Koch, 1977), which confirms that our annotators share a consistent understanding of the task taxonomy and factor semantics underlying the benchmark.

3.3.3 Data Distribution

The dataset comprises 270 curated tasks spanning 9 distinct financial factor categories: Causal, Composite, Math, Momentum, Pattern, Reversion, Risk, Volatility, and Volume. Each category contains an equal number of tasks (30 per category), ensuring balanced coverage across factor families. Importantly, the Causal category includes adversarial scenarios that implicitly tempt models to violate temporal causality, serving as stress tests for safety alignment. To rigorously evaluate algorithmic reasoning capabilities, the difficulty distribution is **intentionally skewed**: Level 1 (20%), Level 2 (20%), and Level 3 (60%). This design prioritizes challenging scenarios and focuses the benchmark on differentiating model capabilities at the frontier of quantitative reasoning, rather than saturating performance on trivial cases.

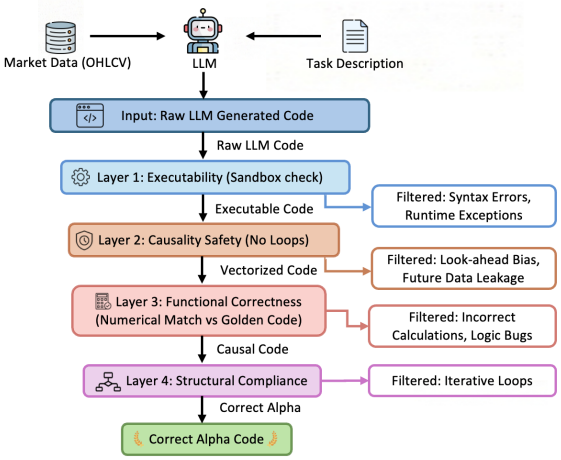

Figure 1: Overview of the multi-layer evaluation protocol in ALPHAQT-BENCH. LLM-generated code is filtered through four diagnostic layers: Executability, Causality Safety, Functional Accuracy, and Structural Compliance, forming a funnel that isolates surface-level success from robust quantitative reasoning.

3.4 Multi-Layer Evaluation Protocol

Standard execution-based metrics (e.g., Pass@1) fail to distinguish valid reasoning from accidental correctness caused by implementation artifacts such as look-ahead bias. As shown in Figure 1, ALPHAQT-BENCH adopts a multi-layer evaluation protocol that progressively enforces execution, causality, functional correctness, and structural compliance, forming a hierarchical diagnostic funnel in which higher-level properties are evaluated only when lower-level constraints are satisfied.

Layer 1: Executability. We first evaluate whether the generated code can be successfully executed without runtime errors. Executability is measured using **Pass@1**, where a generation is considered executable if it runs to completion in a sandboxed environment on both synthetic time-series data and real market data, without raising execution-level exceptions (e.g., syntax errors, shape mismatches, or undefined variables).

Layer 2: Causality Safety. Causality safety evaluates whether an executable program respects temporal and causal constraints, i.e., it does not rely on future information unavailable at decision time. We enforce this requirement using a **Dynamic Trunca-**

tion Test: the program is evaluated on both a full timeline (y_{full}) and truncated timelines (y_{trunc}). A causally valid implementation must satisfy

$$y_{\text{full}}[t] \equiv y_{\text{trunc}}[t], \quad \forall t.$$

Any discrepancy indicates look-ahead bias and renders the program causally invalid. We report the proportion of executable programs that satisfy this condition as the **Causality Safety Rate (Safe)**. Future information leakage is diagnosed when executability exceeds causality safety, indicating executable but temporally invalid implementations.

Layer 3: Functional Accuracy. **Functional Accuracy (Acc)** evaluates task-level semantic correctness. For programs that are both executable and causally safe, we compare the model-generated output series against the *Golden Code*. A solution is considered functionally correct if its output is numerically equivalent to the ground truth. In practice, considering numerical instability introduced by floating-point arithmetic, differences in library implementations, and benign variations in vectorized computation, we do not require strict element-wise equality. If the outputs are not exactly matched, we additionally compute both the Pearson correlation and a normalized RMSE (NRMSE) between the two series. A generation is counted as functionally correct if it satisfies either a correlation above a threshold τ_{corr} or an NRMSE below a threshold τ_{nrmse}; otherwise, it is marked incorrect.

Layer 4: Structural Compliance. While functional correctness ensures semantic validity, it does not guarantee adherence to required structural constraints. Thus, we measure the **Structural Compliance Rate (CSR)**, defined as the proportion of generated programs that satisfy all formatting and structural constraints specified in the prompt, including vectorized operations and the absence of prohibited iterative loops (verified via AST analysis), regardless of functional correctness. Finally, we report **Verified Accuracy (VAcc)** as the strictest end-to-end metric. A generation is counted as correct under VAcc only if it simultaneously satisfies executability, causality safety, structural compliance, and functional correctness.

4 Experiments

4.1 Experimental Setup

Models. We evaluate a diverse set of state-of-the-art LLMs, covering both proprietary and open-weights models to assess reasoning across different scales. The proprietary models include Gemini-2.5 (Comanici et al., 2025), GPT-5-Mini (OpenAI, 2025), Claude-4.5-Haiku (Anthropic, 2025), and Grok-4 (xAI, 2025). The open-weights models comprise Llama-3.1 (Dubey et al., 2024), Qwen-2.5 (Qwen et al., 2025), DeepSeek-V3 (DeepSeek-AI et al., 2025), DeepSeek-R1 (Guo et al., 2025), and GPT-OSS (OpenAI et al., 2025). Implementation details are provided in Appendix A.

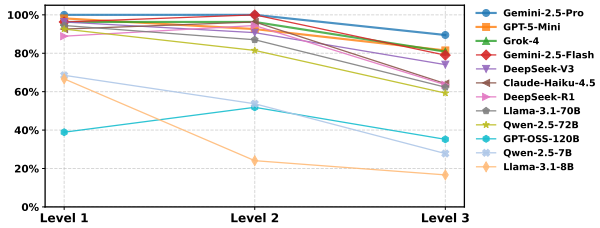

Figure 2: The Complexity Cliff. We compare VAcc across three difficulty levels. While most models achieve high performance on Level 1 (Primitive Grounding), they exhibit sharp degradation at Level 3 (Algorithmic Reasoning).

Metrics. We report results using the proposed multi-layer diagnostic funnel (§ 3.4), including Executability (Pass@1), Causality Safety Rate (Safe), Functional Accuracy (Acc), and Verified Accuracy (VAcc) with Structural Compliance.

4.2 Main Results Analysis

Overall Performance Landscape. Table 2 presents the performance of 12 LLMs across the three difficulty levels of ALPHAQT-BENCH. We could observe Gemini-2.5-Pro establishes the state-of-the-art, achieving an average VAcc of 93.7%. It is the only model that maintains near-perfect performance across all levels, demonstrating a robust mastery of both quantitative logic and vectorization constraints. Among open-weights models, DeepSeek-V3 leads with an average VAcc of 81.9%. However, a clear boundary remains between top-tier proprietary models and open-weights counterparts. The best proprietary model outperforms the best open model by +11.8% in VAcc, suggesting that the capability to handle complex, stateful quantitative constraints is currently a distinguishing feature of frontier-scale models. A category-wise analysis across nine categories is provided in Appendix B.

The Complexity Cliff. While most models achieve near-saturation performance on Level 1

Model	Level 1				Level 2				Level 3				Average			
	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc
GPT-5-Mini	100.0	100.0	98.1	98.1	98.1	98.1	94.4	92.5	97.5	97.5	82.7	81.5	98.1	98.1	88.1	87.0
GPT-OSS-120B	38.9	38.9	38.9	38.9	53.7	53.7	51.9	51.9	43.8	43.8	35.2	35.2	44.8	44.8	39.3	39.3
Gemini-2.5-Flash	100.0	100.0	96.3	96.3	100.0	100.0	100.0	**100.0**	95.1	95.1	81.5	79.1	97.0	97.0	88.1	86.6
Gemini-2.5-Pro	100.0	100.0	100.0	**100.0**	100.0	100.0	100.0	**100.0**	98.8	98.8	89.5	**89.5**	99.3	99.3	93.7	**93.7**
Claude-Haiku-4.5	96.3	94.4	92.6	92.6	100.0	100.0	98.1	96.2	98.8	96.9	79.0	64.1	98.5	97.0	85.6	76.4
Grok-4	98.1	98.1	96.3	96.3	100.0	98.1	96.3	96.3	96.9	96.9	81.5	80.8	97.8	97.4	87.4	87.1
Llama-3.1-8B-It	77.8	77.8	75.9	66.6	33.3	31.5	24.1	24.1	40.7	38.9	17.3	16.7	46.7	45.2	30.4	28.2
Llama-3.1-70B-It	100.0	98.1	94.4	94.4	96.3	96.3	87.0	87.0	87.0	85.2	62.3	62.3	91.5	90.0	73.7	73.7
Qwen-2.5-7B-It	98.1	96.3	90.7	68.5	87.0	81.5	68.5	53.7	74.7	69.1	34.0	27.8	81.9	77.0	52.2	41.1
Qwen-2.5-72B-It	92.6	92.6	92.6	92.6	88.9	87.0	81.5	81.5	84.0	77.8	59.3	59.3	86.7	82.6	70.4	70.4
DeepSeek-V3	98.1	98.1	96.3	96.3	100.0	96.3	90.7	90.7	93.8	92.6	74.1	74.1	95.9	94.4	81.9	81.9
DeepSeek-R1	96.3	94.4	88.9	88.9	96.3	96.3	94.4	94.4	75.3	75.3	63.6	63.6	83.7	83.3	74.8	74.8

Table 2: Main Results on ALPHAQT-BENCH. Best results in terms of VAcc are highlighted in **boldface**. Average denotes task-level performance aggregated across all difficulty levels, rather than a simple mean over three levels. Note that Acc and VAcc are independently reported metrics: a loop-based solution that is numerically correct receives full credit under Acc and is not penalized; VAcc additionally requires structural compliance.

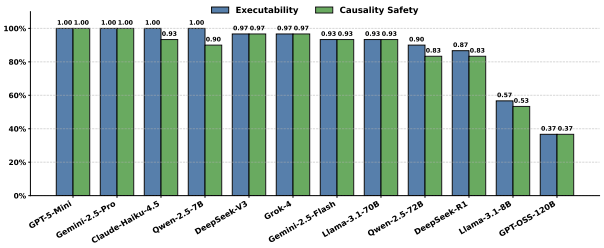

Figure 3: Executability and causality safety on the Causal factor category. A gap between the two metrics indicates future information leakage, where executable implementations rely on unavailable future data.

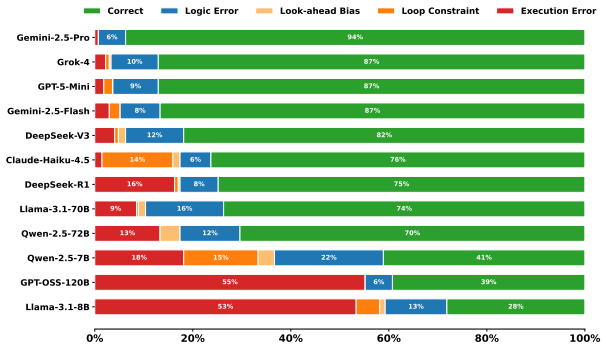

Figure 4: Distribution of failure modes in LLM-generated factors. Errors are categorized into logic errors, look-ahead bias, loop constraint violations, and execution errors, revealing a hierarchical shift in dominant failure modes across model scales.

(Primitive Grounding) and Level 2 (Compositional Semantics), we observe a sharp performance degradation on Level 3 (Algorithmic Reasoning) from Figure 2. For instance, Claude-Haiku-4.5 drops from a VAcc of 96.2% on Level 2 to just 64.1% on Level 3. This cliff indicates that current LLMs struggle significantly when required to translate abstract, stateful financial logic (e.g., recursive slopes or regime-switching) into strict vectorized code, confirming that Level 3 serves as the primary discriminator for expert-level capability.

Leakage Diagnosis in Causal Category. Figure 3 analyzes model behavior on the Causal category by jointly reporting Executability and Causality Safety. Future information leakage is identified when executability exceeds causality safety, indicating that some runnable factor implementations violate temporal or causal constraints. We observe that approximately half of the evaluated models exhibit such a discrepancy, revealing non-negligible leakage despite high executability. Notably, models such as Claude-Haiku-4.5, DeepSeek-R1, and Qwen-2.5-7B generate a substantial fraction of runnable factors that rely on future information. In contrast, Gemini-2.5-Pro, Grok-4, and GPT-5-Mini show identical executability and causality safety scores, indicating strict causal adherence once code is executable. These results demonstrate that executability alone is insufficient for causal validity, and that measuring the executability–causality gap is essential for detecting subtle but critical failure modes in LLM-based factor generation.

Vectorization Constraint: Deployability over Style. The constraint reflects a production-grade requirement, not a stylistic preference. To empirically quantify the performance gap, we conducted controlled runtime experiments on a machine equipped with dual Intel Xeon Gold 6326 CPUs and 256GB RAM, running Ubuntu 22.04. We used a 5,000-row dataset simulating approximately 20 years of daily data for a single stock. For each task, we compared a loop-based implementation generated by Claude-4.5-Haiku, a vectorized implementation generated by Grok-4, and Golden Code. All implementations were executed under

Table 3: Performance comparison of loop, vectorized, and golden implementations

Task	Description	Loop (LLM)	Vectorized (LLM)	Golden	Slowdown vs Vec	Slowdown vs Golden
MOM_HARD_010	Rolling Autocorrelation (20d)	614.88 ± 126.85 ms	1.00 ± 0.27 ms	0.87 ± 0.01 ms	614.7×	706.7×
CAUS_HARD_017	Expanding Beta	2058.66 ± 50.10 ms	1.05 ± 0.15 ms	0.93 ± 0.02 ms	1961.3×	2208.8×

identical hardware and software conditions. The averaged results (mean ± standard deviation over 10 repeated runs) are shown in Table 3. We observe loop-based implementations are 600×–2000× slower than vectorized equivalents under identical hardware conditions. While such latency may appear tolerable in isolated single-asset experiments, it becomes prohibitive in realistic quantitative research pipelines, where backtests often span thousands of assets and multi-year histories. In such settings, the computational cost scales multiplicatively across assets and time windows, rendering loop-based implementations practically infeasible regardless of their numerical correctness. VAcc therefore measures deployability, not style.

4.3 Analysis of Failure Modes

4.3.1 Failure Distribution and Hierarchy

Figure 4 presents a fine-grained decomposition of failure modes in LLM-generated trading factors. Each failure mode corresponds to a specific layer in the diagnostic funnel: execution errors (Layer 1), look-ahead bias (Layer 2), logic errors (Layer 3), and loop constraint violations (Layer 4). Rather than being uniformly distributed, errors exhibit a clear hierarchical structure across model scales.

High-capability models (e.g., Gemini-2.5-Pro, GPT-5-Mini) achieve high correctness rates, with failures dominated by logic errors, indicating challenges at the semantic reasoning level rather than structural compliance or executability. These models largely internalize formatting and constraint requirements, shifting the bottleneck toward complex financial reasoning. Mid-tier models show a more heterogeneous error profile, where look-ahead bias and constraint violations emerge alongside logic errors. This reflects a tension between expressing economically plausible causal relationships and strictly adhering to temporal constraints, consistent with our causality safety analysis. In contrast, lower-capability models are overwhelmingly affected by execution errors and structural violations, with correctness dropping below 40%. For these models, failures occur primarily at the syntactic and structural level, preventing reliable evaluation of higher-level reasoning altogether.

Overall, the results reveal a progressive failure hierarchy: as model capability decreases, errors shift from semantic reasoning failures to constraint violations and finally to execution-level breakdowns. This hierarchy underscores the necessity of multi-dimensional evaluation beyond aggregate accuracy when assessing LLMs for quantitative code generation. Detailed **structural compliance statistics** are reported in Appendix C.

4.3.2 Qualitative Failure Diagnosis

To better understand the cognitive gap between code generation and financial reasoning, we conducted a manual audit of 50 failure cases across the top-performing models (DeepSeek-V3, Llama-3.1-70B, and GPT-5-Mini). We categorized these failures into three distinct reasoning pathologies: *Safety Over-Alignment*, *The Complexity Trap*, and *Semantic Misalignment*.

Safety Over-Alignment: The "Defensive Hallucination". A surprising finding is that Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022) for general safety appears to induce domain-specific reasoning errors. Models frequently conflate calculating a historical feature (state estimation) with executing a trade (action). Consequently, they defensively introduce time lags (e.g., ‘shift(1)‘) that are mathematically incorrect for the requested indicator. As shown in Table 4, Llama-3.1-70B explicitly comments `"Shift returns to avoid look-ahead bias"`. However, the task requests a descriptive statistic of the *current* 20-day window. By shifting, the model evaluates $t-1$ instead of t, destroying the temporal alignment required for feature engineering. This behavior was observed in around 15% of failure cases in Llama-3.1-70B and DeepSeek-V3, particularly in the Risk and Reversion categories.

The Complexity Trap: Over-Engineering Vectorization. While models adhere strictly to the "no-loop" constraint, they often fall into a *complexity trap*. Instead of using standard library functions (e.g., ‘pandas.rolling‘), models attempt to derive complex statistical properties from first principles using low-level NumPy operations (strides, cumula-

Task (RISK_L1_006): Calculate the Percentage of Positive Days ($Close_t > Open_t$) in the last 20 days.

```
# Calculate daily returns
daily_returns = df['day_close'].pct_change()

# Shift returns to avoid look-ahead bias
# ERROR: This shift uses yesterday's data for today's metric
daily_returns = daily_returns.shift(1)

# Create a mask for positive days
positive_days = daily_returns > 0
```

Table 4: Example of Safety Over-Alignment. The Llama-3.1-70B model hallucinates a constraint to "avoid look-ahead bias" for a descriptive statistic, rendering the calculation semantically incorrect.

Task (RISK_L2_017): Calculate K-Ratio (Slope of Log-Equity / StdErr of Slope).

```
# ... 40 lines of setup ...
# Cumulative sums for manual OLS
Sx = x_vals.cumsum()
Sxx = (x_vals**2).cumsum()
Sxy = (x_vals * y).cumsum()

# Manually computing Slope and Intercept per day
# ERROR: Fails to handle NaN/Inf in 'denominator' correctly
denominator = n * Sxx - Sx**2
slope = (n * Sxy - Sx * Sy) / denominator.replace(0, np.nan)
```

Table 5: The GPT-5-Mini model attempts a manual, vectorized implementation of OLS regression using cumulative sums instead of using `rolling().apply()`. This approach resulted in numerical instability and NaNs.

tive sums). This over-engineering increases the surface area for edge-case bugs. In Table 5, the GPT-5-Mini model implements a rolling OLS regression manually using cumulative sum tricks ($O(N)$) rather than a sliding window ($O(N \cdot W)$). While computationally efficient, the implementation fails to handle floating-point stability and 'NaN' propagation correctly, resulting in incorrect outputs.

Semantic Misalignment in Stateful Logic. The most persistent failure mode (accounting for nearly 60% of Level 3 errors) is the inability to faithfully translate stateful financial logic into code. This is particularly evident in recursive indicators like the *Relative Strength Index (RSI)* or *Money Flow Index*, where the definition of "initial state" vs. "steady state" is crucial. For example, in Task `MOM_HARD_001` (Volume-Adjusted RSI), DeepSeek-V3 correctly identified the need for Wilder's Smoothing. However, it attempted to mix vectorized pandas operations with an explicit Python loop:

```
# DeepSeek-V3 Output Fragment
for i in range(14, len(df)):
    avg_gain.iloc[i] = (avg_gain.iloc[i-1]*13 + gain.iloc[i])/14
```

While this logic is mathematically sound, it violates the benchmark's vectorization constraint. In contrast, models that strictly followed vectorization (e.g., using 'ewm(alpha=1/14)') often hallucinated the adjustment factor parameter ('adjust=True' vs 'False'), leading to significant numerical divergence from the Golden Code.

In summary, our qualitative analysis suggests that current LLMs operate as *approximate retrievers* rather than *reasoning engines* in the financial domain. While models often recover the high-level structure of an indicator (e.g., "RSI uses smoothing"), they fail to reason through the strict causal, structural, and numerical implications of domain-specific constraints, leading to subtle but critical errors in alpha generation.

5 Conclusion

In this work, we introduce **AlphaQT-Bench**, a diagnostic benchmark that shifts the evaluation of financial code generation from outcome-driven profitability to *process-level reliability*. By explicitly auditing executability, temporal causality, functional correctness, and structural compliance, AlphaQT-Bench provides a fine-grained diagnosis of reasoning fidelity in financial code generation. Our empirical results reveal a persistent gap between surface-level code generation success and quantitative reasoning: while modern LLMs frequently produce executable implementations, many fail to consistently satisfy causal or structural constraints as task complexity increases. This finding underscores the need for evaluation frameworks that directly assess causal and structural reliability in LLM-based alpha mining systems.

Acknowledgment

This work is funded by China Mobile – HKU Joint Innovation Centre (R24113J4, R26110S3).

Limitations

AlphaQT-Bench has several limitations that suggest directions for future work. First, the benchmark focuses on Python-based implementations, and does not cover other programming lan-

guages or high-performance production environments. Second, models are evaluated in a single-turn generation setting, whereas real-world quantitative development often involves iterative debugging and refinement. Third, although functional correctness is verified numerically, our evaluation may not capture all semantically valid alternative implementations for tasks with inherently ambiguous financial definitions.

Ethical Considerations

This work studies the reliability of LLM-generated financial code and does not evaluate or promote live trading strategies. Performance on ALPHAQT-BENCH should not be interpreted as an indicator of real-world financial profitability, and any generated code should be subject to careful human review before downstream use.

The benchmark is designed to expose potential failure modes such as temporal leakage and structural violations, encouraging more cautious and transparent use of LLMs in quantitative research.

AI-assisted tools, such as ChatGPT, were used for language editing and proofreading. All content were verified and finalized by the authors.

References

Anthropic. 2025. Claude haiku 4.5 system card.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

David H Bailey and Marcos López de Prado. 2014. The deflated sharpe ratio: Correcting for selection bias, backtest overfitting and non-normality. *Journal of Portfolio Management*, 40(5):94–107.

Lang Cao, Zekun Xi, Long Liao, Ziwei Yang, and Zheng Cao. 2025. Chain-of-alpha: Unleashing the power of large language models for alpha mining in quantitative trading. *arXiv preprint arXiv:2508.06312*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Marcos Lopez De Prado. 2018. *Advances in financial machine learning*. John Wiley & Sons.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z.

Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. Deepseek-v3 technical report.

Hongjun Ding, Binqi Chen, Jinsheng Huang, Taian Guo, Zhengyang Mao, Guoyi Shao, Lutong Zou, Luchen Liu, and Ming Zhang. 2025. Alphaeval: A comprehensive and efficient evaluation framework for formula alpha mining. *arXiv preprint arXiv:2508.13174*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Eugene F Fama and Kenneth R French. 1993. Common risk factors in the returns on stocks and bonds. *Journal of financial economics*, 33(1):3–56.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

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*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

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

Fengyuan Liu, Huang Yi, Sichun Luo, Yuqi Wang, Yazheng Yang, Xinye Li, Zefa Hu, Junlan Feng, and Qi Liu. 2025. Cognitive alpha mining via llm-driven code-based evolution. *arXiv preprint arXiv:2511.18850*.

Haochen Luo, Ho Tin Ko, Jiandong Chen, David Sun, Yuan Zhang, and Chen Liu. 2026. Alphabench: Benchmarking large language models in formulaic alpha factor mining. In *The Fourteenth International Conference on Learning Representations*.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b model card.

OpenAI. 2025. Introducing gpt-5.2.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji

Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report.

Weizhe Ren, Yichen Qin, and Yang Li. 2024. Alpha mining and enhancing via warm start genetic programming for quantitative investment. *arXiv preprint arXiv:2412.00896*.

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. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 2813–2822.

xAI. 2025. Grok 4 model card. https://data.x.ai/2025-08-20-grok-4-model-card.pdf.

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.

Junjie Zhang, Shuoling Liu, Tongzhe Zhang, and Yuchen Shi. 2025. A survey on large language model-based alpha mining. *Frontiers of Information Technology & Electronic Engineering*, 26(10):1809–1821.

A Implementation Details

All experiments are conducted using greedy decoding with temperature set to 0.0 to ensure deterministic and reproducible outputs. The complete system prompt and instruction template are provided in Figure 7. Models are accessed through via the `OpenRouter`[1] platform. To isolate semantic and causal errors from market noise, all generated code is executed in a sandboxed Python environment using both synthetic market data and real equity data. Synthetic time series are generated via Geometric Brownian Motion (GBM) to precisely control statistical properties such as volatility regimes, while real data are drawn from individual stocks in the S&P 500 universe. Generated programs must produce valid outputs under *both* settings to be considered executable. To implement the Dynamic Truncation Test, we evaluate each executable program on both the full input sequence and multiple truncated prefixes considered as independent timelines. In practice, we perform this test using five truncations per program to balance coverage and computational efficiency. For each truncation, we compare the outputs on the overlapping time indices and require exact equality. Any discrepancy is treated as a causality violation. Besides, we set $\tau_{\text{corr}} = 0.999$ and $\tau_{\text{nrmse}} = 0.001$ by default. These thresholds are chosen conservatively to tolerate minor numerical differences introduced by floating-point arithmetic and library implementations, while still enforcing strict semantic equivalence.

[1] https://openrouter.ai/

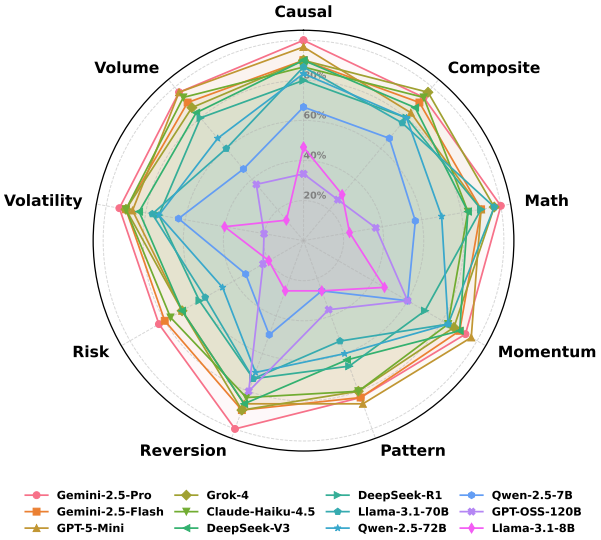

Figure 5: Category-wise comparison of LLM-generated factors across nine dimensions. This illustrates the variance in model expertise, highlighting that recursive domains like *Pattern* are significantly harder to implement correctly than statistical domains like *Volatility*.

B Category-wise Comparison

Figure 5 illustrates a category-wise comparison of different LLMs across nine factor categories, covering both traditional quantitative factor fami-

Model	$\tau_{\text{corr}} = 0.9999$		$\tau_{\text{corr}} = 0.999$		$\tau_{\text{corr}} = 0.99$		$\tau_{\text{corr}} = 0.9$	
	Acc	VAcc	Acc	VAcc	Acc	VAcc	Acc	VAcc
Gemini-2.5-Pro	92.6	92.6	93.7	93.7	94.8	94.8	95.2	95.2
Gemini-2.5-Flash	87.0	85.9	88.1	86.6	89.6	88.1	90.7	89.2
GPT-5-Mini	86.3	85.2	88.1	87.0	90.4	89.3	95.2	93.7
Grok-4	86.3	86.0	87.4	87.1	90.0	89.6	92.2	91.8
Claude-4.5-Haiku	84.8	75.9	85.6	76.4	88.1	77.4	90.7	78.8
DeepSeek-V3	80.0	80.0	81.9	81.9	85.2	84.9	86.3	86.0
DeepSeek-R1	72.6	72.6	74.8	74.8	76.3	75.9	78.1	77.7
LLaMA-3.1-70B	73.0	73.0	73.7	73.7	75.6	75.6	78.1	77.7
Qwen2.5-72B	69.3	69.3	70.4	70.4	70.7	70.7	72.2	72.2
Qwen2.5-7B	51.5	40.4	52.2	41.1	53.7	42.6	56.7	44.5
GPT-OSS-120B	38.9	38.9	39.3	39.3	40.0	40.0	41.9	41.9
LLaMA-3.1-8B	30.4	28.2	30.4	28.2	31.1	28.9	31.9	29.7

Table 6: Performance comparison across models under different τ_{corr} thresholds.

Model	Mean (ms)	Std (ms)	Min (ms)	Max (ms)	N_{Success}
Qwen-2.5-72B-Instruct	2.382	9.108	0.125	83.635	234
DeepSeek-R1	2.574	14.425	0.125	198.594	226
GPT-OSS-120B	3.481	12.561	0.180	93.749	121
Qwen-2.5-7B-Instruct	3.675	19.131	0.128	185.542	224
GPT-5-Mini	3.887	15.490	0.243	158.516	265
LLaMA-3.1-8B-Instruct	5.835	27.200	0.125	278.290	126
Gemini-2.5-Pro	6.545	36.991	0.130	388.044	268
Gemini-2.5-Flash	9.087	38.671	0.132	431.449	262
LLaMA-3.1-70B-Instruct	9.414	35.123	0.125	326.752	247
Grok-4	11.017	42.029	0.128	417.991	264
DeepSeek-V3	12.322	55.524	0.126	588.196	259
Claude-4.5-Haiku	12.693	38.062	0.127	264.491	266

Table 7: Latency statistics (in milliseconds) across models.

lies and reasoning-intensive dimensions. We observe that large, reasoning-oriented models such as Gemini-2.5-Pro exhibit consistently strong and balanced coverage across nearly all categories, indicating robust generalization across heterogeneous factor types. In particular, these models demonstrate clear advantages in the Causal, Risk, and Composite categories, which require integrating economic intuition, multi-step reasoning, and cross-feature interactions. This suggests that stronger language understanding enables models to move beyond surface-level statistical regularities toward more structurally grounded factor construction.

By contrast, smaller models (e.g., Llama-3.1-8B and Qwen-2.5-7B) show more uneven category profiles. While they achieve competitive performance on structurally explicit categories such as Math, Momentum, and Reversion, their performance degrades substantially on categories that demand higher-level abstraction and causal reasoning. Across models, Momentum and Reversion emerge as relatively accessible categories, whereas Causal and Composite exhibit the largest performance dispersion. This pattern underscores a persistent challenge for LLM-based factor discovery: generating causally meaningful and compositionally complex signals remains tightly coupled to model scale and reasoning capacity.

Overall, this analysis indicates that advances in LLM reasoning translate not only into higher average performance but also into broader and more balanced category coverage, which is critical for diversified and robust alpha discovery.

C Structural Compliance Analysis

Figure 6 shows the Structural Compliance Rate (CSR) under two conditions: CSR (Executable), measuring compliance among runnable code, and CSR (Correct), measuring compliance among causally valid implementations. Overall, the results exhibit a clear three-tier distribution across models, revealing distinct behaviors. Approximately one-third of the evaluated models achieve perfect compliance, with CSR values at 1.00 under both executable and correct settings, indicating strong internalization of structural constraints. Another one-third of models exhibit slightly degraded but still high compliance (CSR $\approx$ 0.98–0.99), where

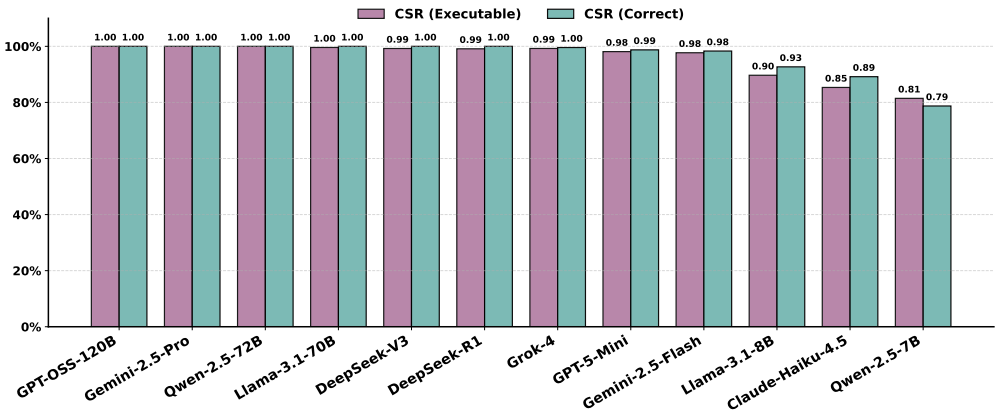

Figure 6: Structural Compliance Rate (CSR) on executable and causally correct code.

occasional violations such as minor formatting inconsistencies or edge-case constraint failures occur but remain sparse rather than systematic. The remaining models show substantially lower CSR, with values dropping to the 0.80–0.90 range, and are characterized by structural instability, including frequent deviations from the required format, missing components, or inconsistent function definitions, making structural errors a dominant failure mode even when the generated code is executable.

Comparing CSR (Executable) and CSR (Correct), we find that structural compliance generally decreases when restricting to causally correct code, indicating that structural violations and causal errors often co-occur. This coupling suggests that models struggling to internalize structural constraints are also more likely to violate higher-level semantic requirements. Overall, this analysis highlights structural compliance as a bottleneck orthogonal to executability, underscoring the importance of evaluating and enforcing structural constraints in LLM-based quantitative code generation.

D Efficiency Analysis

We measured execution time of successfully generated implementations across 10 runs per task under identical hardware conditions (dual Intel Xeon Gold 6326, 64 threads, 256GB RAM). N_Success denotes the number of tasks with valid, executable outputs for each model. The results are shown in Table 7.

Based on these results, we observe three patterns that deepen our qualitative analysis. First, lower N_Success values for GPT-OSS-120B (121) and LLaMA-3.1-8B (126) reflect high execution failure rates, introducing selection bias into their reported runtimes. Their apparently competitive mean latencies are an artifact of surviving only on simpler tasks. Second, higher-capability models (e.g., Gemini-2.5-Pro: 6.545ms) exhibit systematically higher mean runtimes than lower-capability models (e.g., Qwen-2.5-72B: 2.382ms). This reflects a qualitative difference in implementation strategy: stronger models more frequently attempt complex, stateful implementations that are more computationally intensive, while weaker models tend to generate simpler approximations that execute faster but are less semantically correct. Third, the large standard deviations relative to means (e.g., DeepSeek-V3: mean 12.322ms, std 55.524ms) indicate that a small number of generated programs incur disproportionately high execution costs, consistent with our finding that loop-based implementations are 600×–2000× slower than vectorized counterparts.

E Hyperparameter Study

To empirically validate our default parameter choices, we conducted sensitivity analyses across all 12 evaluated models. Table 6 reports threshold sensitivity over τ_{corr} with τ_{nrmse} fixed at 0.001.

Three findings emerge. First, scores are completely stable across the full range of $\tau_{\text{nrmse}} \in [10^{-5}, 10^{-2}]$: all models produce identical Acc and VAcc at every setting, confirming this threshold is not a sensitive parameter in practice. Second, relaxing τ_{corr} from 0.9999 to 0.9 yields modest score increases, but model rankings remain largely consistent (Spearman rank correlation > 0.95 across all settings), confirming that conclusions about model capabilities and the complexity cliff are robust to threshold choice. Our default $\tau_{\text{corr}} = 0.999$ lies within a stable region balancing strictness and tolerance for benign floating-point variation. Third, increasing N_{trun} from 5 to 20 produces no change

Round	Level 1				Level 2				Level 3				Average			
	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc
Round 1	98.1	98.1	96.3	96.3	100.0	98.1	96.3	96.3	96.9	96.9	81.5	80.9	97.8	97.4	87.4	87.0
Round 2	100.0	100.0	98.1	98.1	100.0	100.0	98.1	98.1	99.4	99.4	91.4	90.7	99.6	99.6	94.1	93.7
Round 3	100.0	100.0	98.1	98.1	100.0	100.0	98.1	98.1	100.0	100.0	93.2	92.6	100.0	100.0	95.2	94.8

Table 8: Multi-round performance across evaluation levels in ALPHAQT-BENCH.

Prompt Style	Level 1				Level 2				Level 3				Average			
	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc	Pass@1	Safe	Acc	VAcc
Original	98.1	98.1	96.3	96.3	100.0	98.1	96.3	96.3	96.9	96.9	81.5	80.8	97.8	97.4	87.4	87.1
Concise	96.3	96.3	92.6	92.6	100.0	100.0	98.1	98.1	96.9	96.9	83.3	77.8	97.4	97.4	88.1	84.8
CoT	96.3	96.3	94.4	94.4	98.1	98.1	96.3	96.3	95.7	95.7	85.2	85.2	96.3	96.3	89.3	89.3

Table 9: Impact of prompt styles on multi-level evaluation performance in ALPHAQT-BENCH.

in any model's scores (identical to the default column of Table 6), indicating that five truncations are sufficient to reliably detect all look-ahead violations. Together, these results confirm that our default parameter choices are neither arbitrary nor fragile.

F Multi-round Iterative Evaluation

We conducted additional multi-round iterative evaluation experiments using Grok-4 as a representative model. Specifically, we implemented a feedback-driven pipeline in which the model receives execution error messages and is allowed to regenerate the alpha factor code for up to 3 rounds. The averaged results are summarized in Table 8.

The results demonstrate consistent improvement across all metrics over successive rounds. Notably, Level 3 VAcc improves from 80.9% to 92.6% over three rounds, confirming that the benchmark captures meaningful iterative debugging dynamics. These findings demonstrate that AlphaQT-Bench can be naturally extended to support multi-turn agentic evaluation settings beyond the single-turn baseline reported in the main paper. We note that the single-turn setting is retained as the primary evaluation protocol to ensure controlled, reproducible comparisons across all 12 models. The multi-round results serve as a complementary analysis illustrating the benchmark's flexibility.

G Analysis of Prompt Styles

We conducted a systematic prompt sensitivity analysis using Grok-4 as a representative across three distinct prompt styles: (1) Original. Our default structured prompt with full constraints specified; (2) Concise. A minimal prompt retaining only the task description without any constrains; and (3) CoT. A chain-of-thought prompt explicitly instructing the model to reason step-by-step before generating code. Results are summarized in Table 9.

The results reveal two key findings. First, overall score variance is small. Across all three prompt styles, the range of Average VAcc is 87.1% $\rightarrow$ 89.3%, a spread of only 2.2 percentage points. This demonstrates that AlphaQT-Bench's multilayer evaluation is robust to prompt variation at the aggregate level, and that our reported rankings and conclusions are unlikely to be overturned by prompt choice. Second, the relative difficulty ordering across levels is preserved. Across all three prompt styles, Level 3 consistently yields the lowest Acc and VAcc scores, while Level 1 and Level 2 remain near-saturated. The "Complexity Cliff" phenomenon is stable regardless of prompt formulation.

Notably, CoT prompting yields a modest improvement on Level 3 VAcc (80.8% $\rightarrow$ 85.2%), suggesting that explicit reasoning steps help with algorithmic tasks. This is consistent with prior literature and reinforces our finding that Level 3 tasks genuinely stress-test reasoning capacity rather than surface-level pattern recall.

System Prompt

Role: You are an expert quantitative researcher and Python developer. Your goal is to translate the user's natural language request into a strictly executable Python function for financial alpha factor calculation.

---

1. Data Schema

- Input: A pandas DataFrame `df` representing a single stock's daily historical data.
- Index: `pd.DatetimeIndex`.
- Columns: Strictly `['day_open', 'day_high', 'day_low', 'day_close', 'day_volume']`.

2. Environment & Libraries

Assume the following imports and libraries are already available: `import pandas as pd`, `import numpy as np`, and the `talib`, `scipy.stats`, and `math` modules.

3. Constraints & Best Practices

1. **Vectorization:** Use pandas/numpy vectorized operations. Strictly **NO** `for` loops.
2. **No Look-ahead Bias:** Do NOT use future data to calculate the current day's signal.
3. **Robustness:** Handle `inf` and `NaN` when applicable.
4. **Output Format:** Return a `pd.Series` indexed by `df.index`. The Series name must match the function name.

4. Response Protocol

- **No Markdown:** Provide raw code only (no code blocks).
- **Naming:** Format as `alpha_<category>_<logic>_<details>`.
- **Documentation:** Include a docstring explaining economic intuition.

Example Output:

```python
def alpha_example(df: pd.DataFrame) -> pd.Series:
    '''
    Calculates the 5-day return momentum.
    Logic: (Close_t - Close_t-5) / Close_t-5
    '''
    # Calculation (Note: using day_close as per schema)
    returns = df['day_close'].pct_change(5)

    # Naming and Return
    returns.name = "alpha_example"
    return returns
```

Figure 7: The full system prompt provided to the LLM. It enforces strict data schemas, vectorization constraints, and output formatting to ensure compatibility with the evaluation protocol.