University of Exeter

Department of Mathematics

Interval Forecasting of Cryptocurrency Returns using Quantile Regression Forests: An Application to the Solana Ecosystem

Module Code: MTHM504 – Dissertation

Programme: MSc Applied Data Science and Statistics

James Lewis

27/08/2025

Table of contents

1 Abstract

Cryptocurrency returns are heavy tailed, regime dependent, and poorly served by mean-based forecasts. The problem addressed here is to forecast the distribution of future returns for mid-capitalisation tokens on the Solana blockchain. We study 72-hour log returns for tokens with market capitalisation above $30 million and listing age greater than six months. Raw price, volume, and on-chain metrics are aggregated into 12-hour bars, yielding 6,464 observations between 5 December 2024 and 3 June 2025. The 72-hour target is defined as the difference in log prices six 12-hour bars apart. More than one hundred engineered features span momentum, volatility, market microstructure, on-chain activity, and cross-asset context.

Three models are compared. Linear Quantile Regression (LQR) provides a parametric benchmark. LightGBM uses gradient boosting with the pinball loss to estimate multiple quantiles. The core model is an adapted Quantile Regression Forest (QRF) with time-decay weights, isotonic non-crossing enforcement, and regime-aware residual-quantile offsets, followed by post-hoc calibration. Evaluation follows a blocked rolling cross-validation design with 120 bars for training, a 24-bar calibration gap, and a 6-bar test step (120/24/6), repeated across tokens. Models are assessed by per-quantile pinball loss, empirical coverage of central 80% and 90% intervals, and mean width, with efficiency judged on the coverage–width plane.

Results indicate that QRF achieves the lowest pinball loss in the risk-critical lower tails (tau 0.25) and remains competitive at the median. After calibration, QRF attains near-nominal 90% coverage (0.88) with only modest under-coverage at 80% (0.78). LQR under-covers substantially, while LightGBM over-covers through wider bands. For like-for-like coverage, QRF intervals are materially narrower than LightGBM's, and widths scale appropriately with volatility. When these calibrated intervals drive risk-aware position sizing, a continuous risk-scaled rule achieves a Sharpe of about 0.86 with smaller drawdowns, and a thresholded rule that trades only when the 80% interval is directional reaches a Sharpe of about 0.92. In short, accurate and calibrated interval forecasts translate into tangible trading benefits in the Solana ecosystem.

2 Introduction

2.0.0.1 The Economic Challenge of Forecasting in Emerging Ecosystems

Extreme volatility and non-normal return distributions are defining characteristics of cryptocurrency markets, rendering traditional point forecasts insufficient for robust risk management (Gkillas and Katsiampa, 2018). This challenge is particularly acute for mid-capitalisation tokens within emerging ecosystems like Solana. Unlike large-cap assets, these tokens are subject to rapid narrative shifts and idiosyncratic on-chain dynamics; yet unlike micro-caps, they are liquid enough to attract significant capital. For participants in this space, standard risk models often fail during periods of high network activity or ecosystem-wide events, creating a clear demand for forecasting tools that can dynamically price tail risk.

This dissertation argues that the primary objective must shift from point prediction to interval forecasting. By generating calibrated prediction intervals through conditional quantiles, we can capture the asymmetry and tail risk inherent in these volatile assets. A well-calibrated lower quantile serves as a dynamic, forward-looking analogue to Value-at-Risk (VaR) (Engle and Manganelli, 2004), while the upper quantile informs on potential upside, providing a comprehensive basis for sophisticated risk management and tactical decision-making.

2.0.0.1.1 A Principled Approach: Quantile Regression

To construct reliable prediction intervals, we adopt the framework of quantile regression (Koenker and Bassett, 1978), evaluated with the pinball loss—a strictly proper score for quantiles. We use it throughout for model training and comparison, and provide the formal definition in the **Methodology** and Appendix 2.

This allows models to be evaluated on two critical properties: calibration (does an 80% interval contain the outcome 80% of the time?) and sharpness (are the intervals as narrow as possible while maintaining calibration?) (Gneiting and Raftery, 2007). These properties are paramount in crypto markets, where accurately quantifying both risk and opportunity is the basis of effective strategy.

2.0.0.1.2 The State of the Art and the Research Gap

The literature on cryptocurrency forecasting has largely bifurcated. One branch applies econometric models like GARCH, which excel at modelling volatility but are constrained by parametric assumptions (Bollerslev, 1986). The other employs machine learning, but predominantly for point forecasting of price or direction (Mcnally, Roche and Caton, 2018). While some studies have applied quantile regression to major crypto-assets (Catania and Sandholdt, 2019), they have typically relied on simpler linear models or have not fully leveraged the rich feature set available from on-chain data. Furthermore, the critical step of post-hoc calibration to ensure nominal coverage guarantees, especially for non-parametric methods, is often overlooked.

In doing so, the study prioritises distributional accuracy over point prediction, applies a non-parametric approach to a mid-cap Solana universe, and integrates market microstructure and

on-chain features—choices that, as later results show, yield sharper, better-calibrated intervals than linear and boosted baselines after post-hoc calibration.

2.0.0.2 Problem Formulation: The Solana Ecosystem

This study focuses on forecasting 72-hour log-returns for a universe of mid-cap tokens within the Solana ecosystem, using data aggregated in 12-hour intervals. The asset universe comprises tokens with a market capitalisation exceeding greater than $30 million and listing age greater than 6 months, ensuring a focus on liquid assets. The forecasting model is built upon a feature set designed to capture the multi-faceted drivers of returns, spanning five domains: (i) momentum, (ii) volatility, (iii) market microstructure, (iv) on-chain activity, and (v) cross-asset context.

2.0.0.3 Methodology and Contributions

The central hypothesis is that the non-linear, interaction-heavy nature of this market demands a non-parametric approach. We propose an adapted Quantile Regression Forests (QRF) model (Meinshausen, 2006). QRF was selected as the primary model for several reasons. Firstly, its ensemble nature provides inherent robustness to the noisy predictors common in high-dimensional financial feature sets. Secondly, unlike gradient boosting, QRF's independent tree construction can be less prone to overfitting in non-stationary environments. Finally, its method of estimating quantiles from the full distribution of training samples in terminal nodes is a more direct and empirically stable approach than methods requiring separate models for each quantile. To tailor the model to a non-stationary financial time series, we apply: (i) time-decay sample weights to prioritise recent data; (ii) non-crossing enforcement via isotonic rearrangement across ; and (iii) post-hoc calibration using regime-aware residual-quantile offsets, with split-conformal bands as a robustness check. These adaptations are detailed in the Methodology 4.3–4 and Appendix 2.

Contributions.

1. **Methodology.** A QRF-based interval pipeline for 72-hour returns with time-decay weighting, isotonic non-crossing, and post-hoc calibration (residual-quantile; split-conformal as a check).
2. **Evaluation design.** Rolling, block-wise validation tailored to non-stationary crypto series; comparison to LQR and LightGBM using pinball loss, empirical coverage, and interval width.
3. **Empirical insight.** Feature-family analysis of tail drivers and an applied risk-use case (interval-aware sizing).

We benchmark our adapted QRF against a parametric Linear Quantile Regression (LQR) and a powerful LightGBM model (Ke *et al.*, 2017) augmented with **conformal prediction** (Romano, Patterson and Candès, 2019). Preliminary results suggest that the primary advantage of the adapted QRF framework lies in its superior ability to synthesise on-chain activity and market microstructure features to anticipate shifts in return distribution skewness—a dynamic that linear models fail to capture.

2.0.0.4 Scope and Delimitations

This dissertation provides a rigorous methodological and empirical analysis of interval forecasting. It does not aim to develop a complete, production-ready trading system, which would require further considerations such as transaction costs, liquidity constraints, and execution latency. Furthermore, the feature set, while comprehensive, is confined to publicly available market and on-chain data, thereby excluding alternative data sources such as social media sentiment or developer activity metrics, which may also contain predictive information. The findings are specific to the mid-cap tokens within the Solana ecosystem during the observation period and may not be directly generalisable to other blockchains, market-cap tiers, or market regimes without further investigation and potential recalibration.

2.0.0.5 Research Questions

This framework motivates these core research questions:

1. **RQ1.** Accuracy across quantiles and regimes. Across the $\{0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95\}$ and different volatility regimes, do Quantile Regression Forests (QRF) produce consistently lower pinball loss than Linear Quantile Regression (LQR) and LightGBM baselines?
2. **RQ2.** Calibration and sharpness across tokens. After post-hoc calibration, can QRF intervals maintain near-nominal coverage (80 % and 90 %) while achieving narrower widths than LQR and LightGBM, and does this hold across tokens and regimes?
3. **RQ3.** Trading utility. Does the superior statistical quality—accuracy, calibration and sharpness—of QRF prediction intervals translate into demonstrable economic value when applied to a risk-aware portfolio of mid-cap Solana tokens?

3 Literature Review

3.0.0.1 The Challenge: Statistical Properties of Cryptocurrency Returns

The return distributions of cryptocurrencies are characterised by heavy tails, significant skew, and extreme kurtosis relative to traditional assets, reflecting the frequency of large, abrupt price movements (Gkillas and Katsiampa, 2018). This leptokurtosis is compounded by pronounced volatility clustering, periods of relative calm followed by explosive variability, a dynamic exacerbated by the market's continuous operation and fragmented liquidity, which can amplify shocks across uncoordinated venues.

Crucially, this extreme risk is also largely idiosyncratic to the crypto market. Major cryptocurrencies carry substantial tail risk that is not strongly correlated with traditional stock market indices; instead, extreme events are driven by crypto-specific factors such as investor sentiment, regulatory news, or network-level events (Borri, 2019). Furthermore, their returns show little to no exposure to standard macroeconomic risk factors, being influenced instead by internal drivers like network momentum and adoption metrics (Liu and Tsyvinski, 2021). This body of evidence demonstrates that classical financial risk models, with their reliance on Gaussian assumptions and traditional risk factors, are fundamentally misspecified for crypto assets. A credible forecasting framework must therefore abandon these assumptions and be built to incorporate the crypto-native features that drive risk.

3.0.0.2 Approaches to Quantile Estimation

Given the non-normal character of crypto returns established previously, estimating the full conditional distribution is more informative than forecasting its central tendency. Quantile regression provides a natural framework for this, but the choice between a restrictive parametric model and a flexible non-parametric one is critical.

3.0.0.3 The Parametric Benchmark: Linear Quantile Regression

Quantile regression, introduced by (Koenker and Bassett, 1978), generalises linear regression by estimating conditional quantiles directly. We evaluate quantile models using the pinball loss, a strictly proper scoring rule for quantiles (see 5.4). While LQR provides a transparent and interpretable benchmark, its fundamental assumption of a fixed linear relationship across all quantiles represents a severe limitation. This rigidity is fundamentally at odds with the non-linear volatility dynamics and abrupt regime shifts that define cryptocurrency markets. Furthermore, the common practical issue of quantile crossing, where independently estimated quantile lines intersect, and can yield incoherent and unusable interval forecasts unless post-hoc remedies like rearrangement are applied (Chernozhukov, Fernández-Val and Galichon, 2010). These shortcomings do not merely motivate, but necessitate the exploration of more flexible, non-parametric methods.

3.0.0.4 Non-Parametric Solutions: Quantile Regression Forests

As a direct response to the limitations of linear models, Quantile Regression Forests (QRF), proposed by (Meinshausen, 2006), extend the Random Forest algorithm (Breiman, 2001) to estimate the entire conditional distribution. Instead of averaging outcomes in terminal nodes, QRF uses the full empirical distribution of training responses within the leaves to form a predictive distribution, from which conditional quantiles are derived.

This non-parametric approach is inherently well-suited to financial data; it naturally captures complex non-linearities and adapts to heteroskedasticity without pre-specification. However, QRF is not without its own challenges. Its theoretical foundation rests on an assumption of independent and identically distributed data (i.i.d.), a condition clearly violated by financial time series. A naive application of QRF to time-ordered data can therefore lead to biased estimates. This violation is a central methodological challenge that requires specific adaptations, such as the time-decay weighting and rolling validation schemes discussed later, to apply the model soundly. Furthermore, the accuracy of its tail quantile estimates can degrade if the terminal leaves are sparsely populated, a genuine risk when modelling extreme events. Boosting offers another route to non-parametric quantile estimation, but with contrasting properties.

3.0.0.5 A Boosting Alternative: LightGBM for Quantiles

Gradient boosting presents another powerful non-parametric paradigm. It constructs an ensemble sequentially, with each new tree trained to correct the errors, specifically, the gradients of the loss function—of the preceding models (Friedman, 2001). This methodology can be directly applied to quantile regression by using the pinball loss as the objective. LightGBM (Ke *et al.*, 2017) is a highly efficient and scalable implementation of this idea, making it a formidable baseline.

In sharp contrast to QRF's parallelised construction, boosting's sequential focus on difficult-to-predict instances can yield sharper estimates in the tails. This potential for higher accuracy, however, comes with significant trade-offs. A separate model must typically be trained for each target quantile, imposing a considerable computational burden. The aggressive, error-focused fitting can also produce "ragged" and unstable quantile estimates in regions with sparse data and may overfit transient noise without careful regularisation. Finally, like LQR, independently fitted boosting models are susceptible to the problem of quantile crossing.

3.0.0.6 Ensuring Rigour: Calibration, Evaluation, and Comparison

Selecting a flexible forecasting model is insufficient on its own; its predictive performance must be evaluated using principled metrics, its outputs calibrated to ensure reliability, and its superiority over alternatives established through formal statistical tests.

3.0.0.7 Proper Scoring and Forecast Evaluation

A principled evaluation of probabilistic forecasts requires the use of strictly proper scoring rules, which incentivise the model to report its true belief about the future distribution. For quantile forecasts, the canonical proper scoring rule is the pinball loss (Gneiting and Raftery, 2007). As the metric being directly optimised by the models, it serves as the primary tool for evaluation. However, performance is not a single dimension. The quality of an interval forecast is judged by two distinct and often competing properties: calibration, the statistical consistency between

the nominal coverage rate (e.g., 90%) and the empirical frequency of outcomes falling within the interval; and sharpness, the narrowness of the interval. An ideal forecast is one that is maximally sharp, subject to being well-calibrated. However, a model optimised on a proper score is not inherently guaranteed to be well-calibrated in finite samples. This gap between theoretical optimisation and empirical reliability motivates the use of formal calibration techniques.

3.0.0.8 Achieving "Honest" Intervals: Conformal Prediction

Conformal prediction provides a distribution-free framework to correct for such miscalibration. Specifically, Conformalized Quantile Regression (CQR) (Romano, Patterson and Candès, 2019) provides a mechanism to adjust a base model's quantile forecasts to achieve guaranteed marginal coverage. It uses a hold-out calibration set to compute a conformity score based on model errors, which is then used to adjust the width of future prediction intervals. While the underlying exchangeability assumption is violated in time series, employing a rolling calibration window of recent data provides a practical and widely used compromise to adapt the procedure to non-stationary environments.

3.0.0.9 Statistically Significant Comparisons: The Diebold-Mariano Test

To move beyond descriptive comparisons of average loss, formal statistical tests are required to determine if the performance difference between two models is significant. The Diebold-Mariano (DM) test (Diebold and Mariano, 1995) provides a standard framework for this, on rolling loss differentials (see Methods). For the multi-step, overlapping forecasts used in this project, the sequence of loss differentials will be autocorrelated by construction. It is therefore critical to use a heteroskedasticity and autocorrelation consistent (HAC) variance estimator, as recommended by (West, 1996), to ensure valid statistical inference.

3.0.0.10 Methodological Requirements for Robust Time-Series Forecasting

The foundational literature establishes the potential of non-parametric models, but their successful application to volatile, non-stationary financial time series is contingent upon a number of specific methodological adaptations. This section reviews the literature concerning these essential requirements, from ensuring the logical coherence of predictions to adapting models to the temporal dynamics of the data.

3.0.0.11 Ensuring Coherent Predictions: Non-Crossing Quantiles

Models that estimate quantiles independently, such as LQR and standard gradient boosting, are susceptible to the critical failure of quantile crossing. This occurs when, for instance, a predicted 90th percentile falls below the predicted 50th percentile, yielding an illogically and unusable conditional distribution. To rectify this, (Chernozhukov, Fernández-Val and Galichon, 2010) proposed a post-processing "rearrangement" technique. This method applies isotonic regression to the initially estimated quantile function, projecting the unconstrained predictions onto the space of valid, non-decreasing distribution functions. This ensures the monotonicity of the quantile curve and is a critical step for producing valid prediction intervals (see Methods).

3.0.0.12 Adapting to Non-Stationarity and Temporal Dependence

Financial time series are fundamentally non-stationary and autocorrelated, violating the i.i.d. assumption that underpins many machine learning models. Two distinct but related adaptations are required to address this.

First, to handle non-stationarity such as volatility clustering, the model must prioritise more recent information. The literature supports the use of time-decay sample weights to achieve this. (Taylor, 2008), for example, introduced exponentially weighted quantile regression for Value-at-Risk estimation, demonstrating that up-weighting recent observations yields more responsive and accurate tail forecasts in changing market conditions.

Second, to handle temporal dependence, model evaluation and hyperparameter tuning must respect the chronological order of the data. Standard k-fold cross-validation is invalid for time series, as it can lead to lookahead bias and produce misleadingly optimistic performance estimates. The literature therefore strongly advocates for rolling-origin or blocked cross-validation schemes, which preserve the temporal sequence by training only on past data to forecast the future, thereby simulating a live forecasting environment (Bergmeir, Hyndman and Koo, 2018).

3.0.0.13 Correcting for Bias and Ensuring Empirical Calibration

Even correctly specified quantile models can exhibit systematic biases in finite samples. As (Bai *et al.*, 2021) have shown, linear quantile regression can suffer from a theoretical under-coverage bias, where a nominal 90% interval may contain the true outcome significantly less than 90% of the time due to estimation error. This problem motivates the necessity of post-hoc calibration.

While the CQR framework discussed previously is one such solution, the literature offers several alternatives. Methods like the Jackknife+ (Barber *et al.*, 2021) and residual bootstraps provide different mechanisms for constructing prediction intervals with more reliable coverage properties. The existence of this rich literature on calibration highlights a crucial principle for risk management applications: a model's raw output cannot be taken at face value. An explicit calibration step is required to correct for inherent biases and ensure the resulting prediction intervals are empirically "honest".

3.0.0.14 Integrating Crypto-Native Data Sources

The literature on cryptocurrency risk factors makes it clear that models confined to historical price data are insufficient. The unique nature of blockchain-based assets provides a rich set of crypto-native data sources that are essential for capturing the specific drivers of risk and return in this asset class.

3.0.0.15 Market Microstructure and Liquidity

Like traditional markets, cryptocurrency price dynamics are influenced by liquidity and trading frictions. Empirical studies have documented that periods of market stress coincide with widening bid-ask spreads and evaporating order book depth (Dyhrberg, 2016). Furthermore, the on-chain nature of transactions introduces unique microstructural features, such as network congestion and transaction fees, which can impact market liquidity and price formation (Easley, O'Hara and Basu, 2019). Incorporating proxies for these effects is crucial, as it allows a model to

dynamically adjust its estimate of uncertainty; for instance, by widening its prediction intervals in response to deteriorating market liquidity, thereby anticipating volatility spikes.

3.0.0.16 On-Chain Activity and Network Fundamentals

Blockchains provide a transparent ledger of network activity, offering powerful proxies for an asset's fundamental adoption and utility. Metrics such as the growth in active addresses, on-chain transaction counts, and, in the context of decentralised finance (DeFi), the Total Value Locked (TVL) in smart contracts, can signal shifts in investor sentiment and capital flows. Empirical studies consistently find that models augmented with on-chain metrics significantly outperform those based only on historical prices, as this data provides unique information about network health and demand (Sebastião and Godinho, 2021). These features allow a model to condition its forecasts on the fundamental state of the network, potentially informing not just the location but also the shape of the predictive distribution.

3.0.0.17 Cross-Asset Spillovers and Systemic Risk

The cryptocurrency market is a highly interconnected system where shocks to major assets like Bitcoin and Ethereum often propagate to smaller altcoins. This "connectedness" has been formally measured, showing significant return and, particularly, volatility spillovers from market leaders to the rest of the ecosystem (Diebold and Yilmaz, 2014; Koutmos, 2018). This implies that the risk of an individual token is not purely idiosyncratic; it is also a function of the broader crypto market's state. Consequently, any forecasting model that treats a token in isolation is fundamentally misspecified and is likely to underestimate systemic risk. A robust framework must therefore account for these cross-asset influences.

3.0.0.18 Synthesis and Conclusion

This review has established a clear and compelling justification for the methodology adopted in this dissertation. The unique statistical properties of cryptocurrency returns—heavy tails, volatility clustering, and dependence on idiosyncratic, on-chain factors—render traditional parametric models inadequate. This failure necessitates the use of flexible, non-parametric methods, for which Quantile Regression Forests are a logical choice, given their ability to capture complex, non-linear relationships without restrictive distributional assumptions.

However, the literature also makes it clear that a naive application of any such model would be insufficient. A credible forecasting framework requires a series of specific, evidence-based adaptations. The need to adapt to non-stationarity justifies the use of time-decay weighting. The imperative for valid, coherent predictions necessitates post-processing to enforce non-crossing quantiles. The requirement for reliable out-of-sample evaluation mandates the use of rolling cross-validation. Finally, the well-documented tendency for quantile models to mis-calibrate compels the integration of a formal calibration step to ensure the final prediction intervals are empirically valid.

By synthesising these distinct strands of literature—from model selection to time-series adaptation and calibration—this project constructs an integrated and methodologically robust framework. This framework is specifically designed to address the multifaceted challenges of interval forecasting in the volatile and rapidly evolving cryptocurrency market.

4 Data and Features

The validity of any forecasting model is fundamentally constrained by the quality and integrity of its input data. For volatile and rapidly evolving assets like cryptocurrencies, constructing a robust, research-grade dataset is a critical prerequisite for meaningful analysis. This chapter details the multi-stage process undertaken to source, clean, and engineer the data used in this dissertation. It begins by defining the asset universe and data sources, then describes the construction of the target variable and the extensive feature engineering pipeline. Finally, it outlines the preprocessing steps taken to handle missing data and presents key insights from the exploratory data analysis that guided the modelling approach.

See Appendix 1 and the respective sections within Appendix 4 (Figures) and 5 (Code) for additional supporting content

4.1 Data Sourcing and Asset Universe Definition

We ingested data using a hybrid Python and TypeScript pipeline, with more than ten modular ingestion scripts covering OHLCV, order-book snapshots, and on-chain endpoints, plus two processing notebooks for aggregation and quality checks. *Links to these scripts and notebooks can be found in Appendix 5.*

The foundation of this study is a bespoke, multi-source panel dataset constructed at a 12-hour resolution, specifically designed to support tail-sensitive, quantile-based forecasting. The data streams were aggregated from several high-quality APIs, using specific endpoints for different data types to ensure the highest fidelity for each signal.

The primary data sources included:

- **Price and Volume Data**: Historical Open-High-Low-Close (OHLC) and volume data for individual tokens were sourced from the **SolanaTracker API**, chosen for its deep liquidity and high-quality, reliable price feeds, which minimises the risk of spurious gaps or errors in the core price series.
- **On-Chain Metrics**: Key on-chain indicators for the Solana ecosystem, such as `holder_count` and `transfer_count`, were retrieved via the **CoinGecko API**, which provides broad coverage of token-specific network activity.
- **Global Context Data**: Broader market signals, including historical prices for Bitcoin (BTC) and Ethereum (ETH), as well as Solana-specific network metrics like transaction counts and Total Value Locked (TVL), were sourced from CoinGecko and the **Google BigQuery Solana Community Public Dataset**.

An initial effort was also made to incorporate social media sentiment data, given the narrative-driven nature of many Solana tokens. While an ingestion pipeline was successfully built, the data availability and quality were ultimately deemed insufficient for rigorous academic analysis and were excluded from the final feature set.

The **asset universe** was carefully defined from an initial, hand-picked list of 23 tokens based on their relevance to the Solana ecosystem. This list was then filtered according to a set of

rigorous criteria. To be included in the final universe, a token had to satisfy: *See the full token list in Appendix 1 (Table 1)*

- A **minimum market capitalisation** of $30 million to ensure a baseline of market significance and liquidity.
- A **minimum trading history** of six months. This was a crucial requirement to ensure a stable two-month (approx. 120 12-hour bars) training window was available for the initial backtest period for every asset in the universe.

This filtering was a deliberate and aggressive research design choice. Tokens with excessive missing data, insufficient history, or erratic reporting were pruned from the sample. This step is critical for quantile-based modelling, as tokens with inconsistent or late-starting data histories can inject significant bias into the empirical distribution, particularly distorting the tail estimates that are a primary focus of this research. By favouring data quality over sample size, this process ensures that the subsequent modelling results are attributable to the forecasting methodology itself, rather than being artefacts of poor-quality data.

The initial raw dataset comprised **8,326 rows across 23 tokens**. After applying the filtering criteria and cleaning procedures detailed in the following sections, the final asset universe for this study consists of **20 mid-cap Solana tokens**. The full dataset spans from **5th December 2024 to 3rd June 2025**, comprising a total of **6,464** 12-hour observations after cleaning and alignment. *See Appendix 1 for raw feature dataset snippet (Table 2)*

4.2 Target Variable Construction and Properties

The predictive target for this study is the **72-hour forward-looking logarithmic return**, calculated at each 12-hour time step. It is formally defined as:

$$r_t^{(72h)} = \log(P_{t+6}) - \log(P_t)$$

where P_t is the closing price of the token at the end of the 12-hour bar at time t, and P_{t+6} is the closing price six 12-hour bars later, *see Appendix 1 for code*. The use of logarithmic returns is standard practice in financial econometrics (Campbell, Lo and MacKinlay, 1997), as it provides a continuously compounded return that is time-additive and whose distribution more closely approximates normality than simple returns.

The combination of a **12-hour data cadence** and a **72-hour forecast horizon** was a deliberate design choice to create a target variable suitable for mid-frequency trading strategies. The 12-hour aggregation smooths the extreme noise present in sub-hourly price changes, while the 72-hour horizon is long enough to capture significant, economically meaningful moves where tail events and distributional properties become highly relevant. This choice aims to maximise the signal-to-noise ratio for the specific purpose of forecasting the distribution of multi-day returns.

Exploratory analysis confirms that the resulting target variable exhibits the extreme non-normal characteristics that motivate this research. As shown in **Table 1**, the pooled distribution of 72-hour log returns is highly leptokurtic and positively skewed. With a kurtosis of 20.73, it demonstrates exceptionally fat tails compared to a normal distribution (kurtosis of 3), indicating that extreme price movements are far more common than a Gaussian model would suggest.

Statistic	Value
Mean	0.0031
Standard Deviation	0.1259
Skewness	**1.68**
Kurtosis	**20.73**
Minimum	-0.75
Maximum	1.02

Table 1: Summary Statistics for the 72-hour Log Return Target Variable (Pooled Across All Tokens).

The heavy-tailed nature of the target is further illustrated in **Figure 4.1**, which plots the empirical distribution against a normal distribution with the same mean and variance. The substantially higher peak and elongated tails of the empirical distribution provide clear visual evidence that a Gaussian assumption would be inappropriate and underscore the necessity of a quantile-based modelling approach.

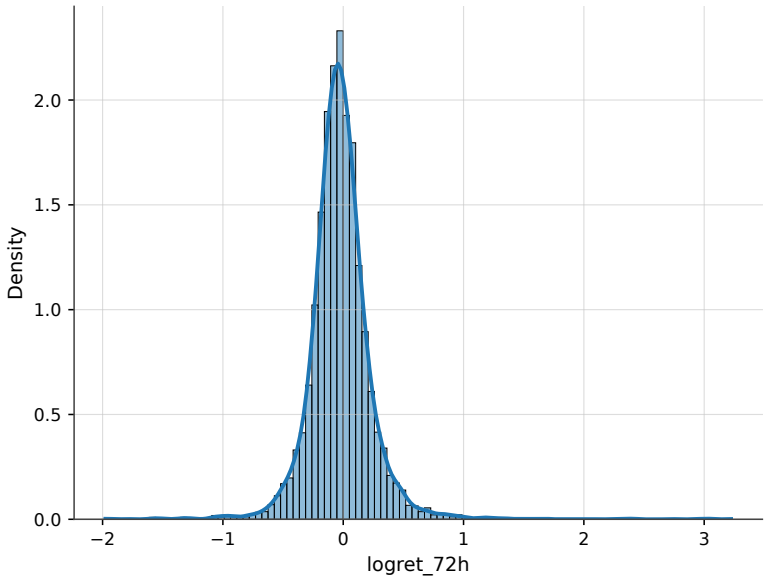


Figure 4.1: Distribution of the 72-hour log return target variable, pooled across all tokens. The distribution exhibits a sharp peak and significantly heavier tails than a comparable normal distribution, justifying the use of quantile-based models.

Furthermore, the properties of this distribution are not static. Analysis of the underlying 12-hour returns reveals that the shape of the distribution is highly conditional on the broader market environment. By defining macro-regimes based on the quartiles of SOL's 12-hour return, it becomes evident that the conditional distribution of token returns shifts systematically. As shown in **Figure 4.2**, "Bull" regimes are associated with positive skew and a fatter right tail, whereas "Bear" regimes exhibit heavier left tails, signalling increased downside risk. This empirical finding of regime-dependence is critical, as it justifies the need for a *conditional* forecasting model that can adapt its predicted quantiles based on contextual features.

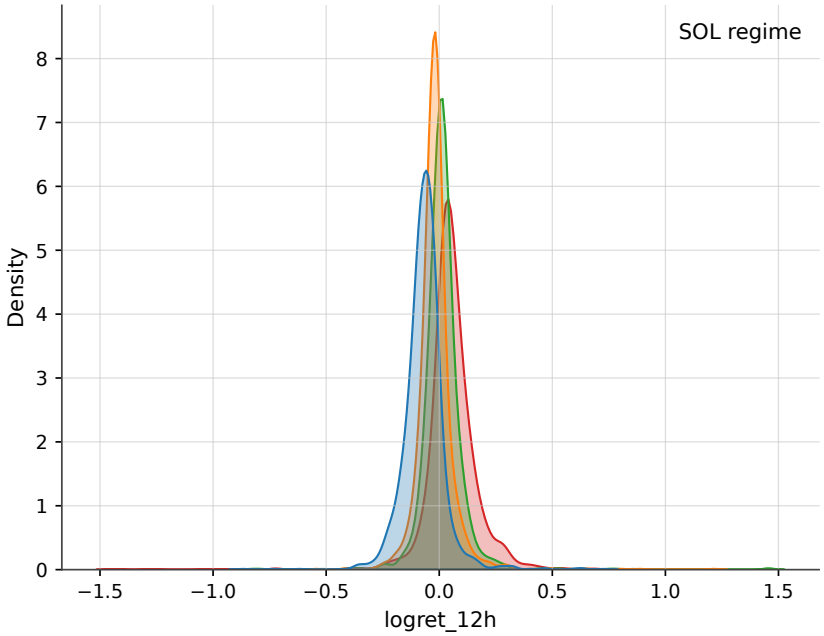


Figure 4.2: Conditional distribution of 12-hour token returns, faceted by the SOL macro-regime. The shape, skew, and tail behaviour of the returns change significantly depending on the broader market context.

Finally, it is important to note that this construction results in an overlapping target variable. A new 72-hour forecast is generated every 12 hours, meaning that the forecast periods overlap significantly. This has direct implications for the evaluation methodology, as the resulting forecast errors will be serially correlated by construction. As detailed in the literature review and methodology chapters, this requires the use of specific techniques, such as blocked cross-validation and HAC-robust statistical tests, to ensure valid inference.

4.3 Data Preprocessing and Cleaning

The **cleaning strategy** proceeded in stages: (i) temporal alignment to fixed 12-hour bins; (ii) gap detection and minimal winsorisation; (iii) removal or flagging of structurally missing segments; and (iv) post-aggregation QA checks. Implementation snippets are referenced in the Appendix.

Before any features could be engineered, the raw, multi-source panel dataset underwent a rigorous cleaning and preprocessing pipeline. This was a critical phase designed to handle the significant data quality challenges inherent in cryptocurrency markets, such as missing observations and inconsistent token histories, ensuring the final dataset was robust and suitable for modelling. The initial raw data contained approximately **18% missing values** in the core OHLCV columns alone, with some on-chain features like `holder_count` missing nearly 40% of their data (see Appendix 1, Table 3 for a full breakdown).

The nature of this missingness was not uniform. As illustrated by the heatmap in **Figure 4.3**, the data gaps were highly structured. Some tokens (e.g., `MEW`, `ZEREBRO`) had clean data but only after a late start date, while others exhibited intermittent, patchy gaps throughout their history. This heterogeneity necessitated a multi-step strategy rather than a single, one-size-fits-all approach.

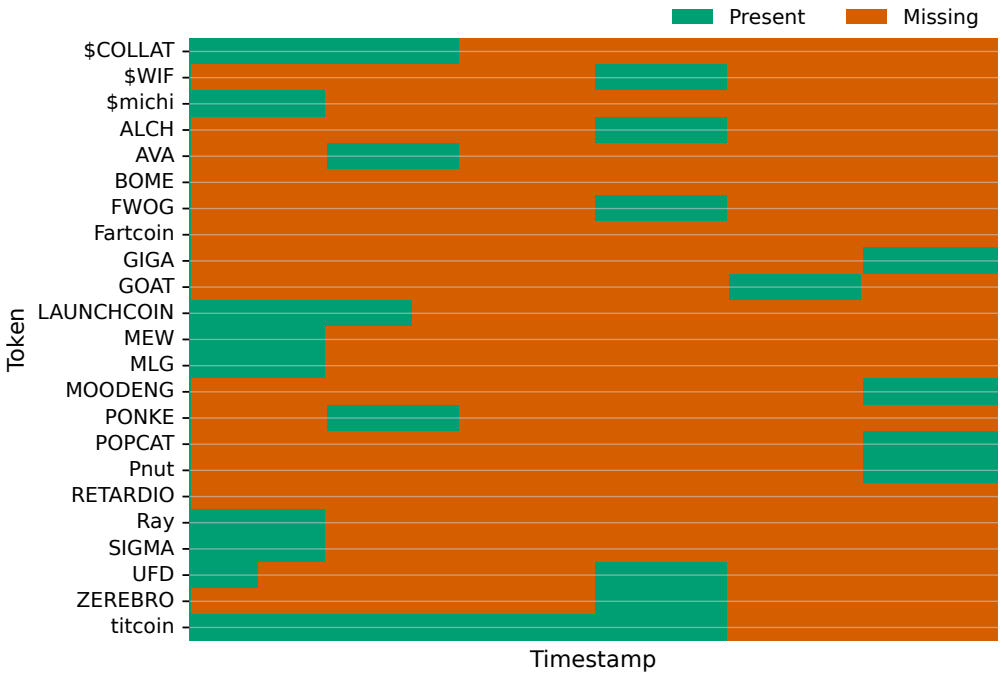


Figure 4.3: Heatmap of OHLCV data presence across the token universe over time (Green = Present, Red = Missing). The block-like structure for some tokens indicates late listings, while sporadic red patches show intermittent data gaps, motivating a hybrid cleaning strategy.

4.3.0.1 Temporal Alignment and Clipping

To ensure temporal consistency, each time series was first clipped to begin only from its first fully valid OHLCV observation. This step, detailed in the strategy summary in Appendix 1 **@sec-cleaning-strategy**, removes spurious data from pre-launch or illiquid initial listing periods, which could otherwise contaminate the analysis. Tokens with insufficient history after this clipping process (e.g., `$COLLAT`) were dropped from the universe entirely.

4.3.0.2 Imputation Strategy

A key challenge was to fill the remaining intermittent gaps without distorting the underlying distributional properties of the data. Several imputation methods were benchmarked on simulated missing data. Counter-intuitively, the analysis revealed that a simple linear interpolation outperformed more complex methods like Kalman smoothing in terms of Root Mean Squared Error (RMSE) (see Appendix 1 Table 4 **@#imp-table** for benchmark results). This finding suggests that for small, sporadic gaps, a simple interpolation preserves the local price trajectory and its inherent noise structure more effectively than methods that impose stronger, potentially smoothing, structural assumptions.

Therefore, the final strategy adopted was a hybrid approach: linear interpolation to fill the majority of gaps, supplemented by a forward-fill for a maximum of two consecutive 12-hour bars.

4.4 Exploratory Data Analysis

Following the preprocessing pipeline, an extensive exploratory data analysis (EDA) was conducted to uncover the key empirical properties of the data. This section presents the three most critical findings that provide a direct, data-driven justification for the subsequent feature engineering choices and the selection of a non-parametric, conditional forecasting model. *See Appendix 4 for full EDA figures, and Appendix 5 code*

4.4.0.1 Volatility Clustering and Asymmetric Leverage

The data exhibits two foundational properties of financial time series that invalidate simple, static risk models. First, strong volatility clustering is evident in the autocorrelation of absolute 12-hour returns, confirming that risk is time-varying and motivating the inclusion of dynamic volatility features.

Second, the relationship between returns and subsequent volatility is asymmetric. An analysis regressing 12-hour log returns against forward 36-hour realised volatility reveals a distinct U-shaped pattern, as shown in **Figure 4.4**. This confirms that variance is conditional on the magnitude of recent returns, with large moves in either direction predicting elevated future volatility. The effect is slightly stronger for negative returns, consistent with a "crypto leverage effect" where downside shocks lead to greater market instability. This non-linear dynamic necessitates a modelling approach, such as the Quantile Regression Forest used in this study, that can naturally capture such relationships and adapt its prediction interval widths based on the direction and magnitude of recent price shocks.

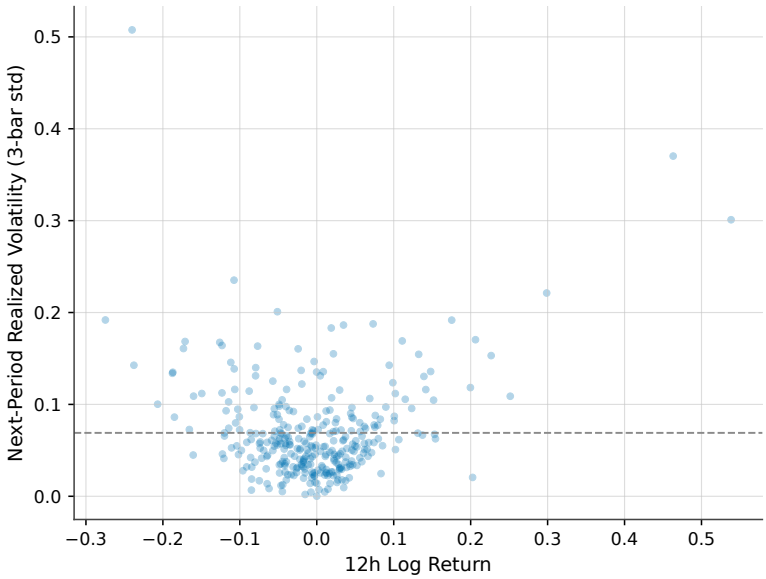


Figure 4.4: A scatter plot of 12h Log Return vs. Next-Period Realised Volatility. The U-shaped pattern, with a slightly steeper slope for negative returns, illustrates the asymmetric leverage effect.

4.4.0.2 Feature Redundancy and Collinearity

An analysis of the correlation structure between 18 core features was conducted to identify potential multicollinearity, which can destabilise tree-based ensemble models. As shown in the

Pearson correlation matrix in **Figure 3.5**, several feature pairs exhibit extremely high linear relationships. The most significant correlations were observed between:

- `token_close_usd` and `token_volume_usd` (r $approx 0.999$)
- `btc_close_usd` and `tvl_usd` (r $approx 0.94$)
- `sol_close_usd` and `tvl_usd` (r $approx 0.89$)

This finding is critical. It implies that highly collinear inputs can inflate variance, degrade interpretability, and reduce generalisation; we therefore prioritise redundancy control (filtering/aggregation) during feature engineering and in later pruning. While these raw fields were retained for the initial feature engineering phase to allow for the construction of richer indicators (e.g., from OHLC data), this analysis motivates the necessity of a subsequent feature pruning step. Reducing this high level of collinearity before modelling is essential for improving the stability, training speed, and interpretability of the final Quantile Regression Forest.

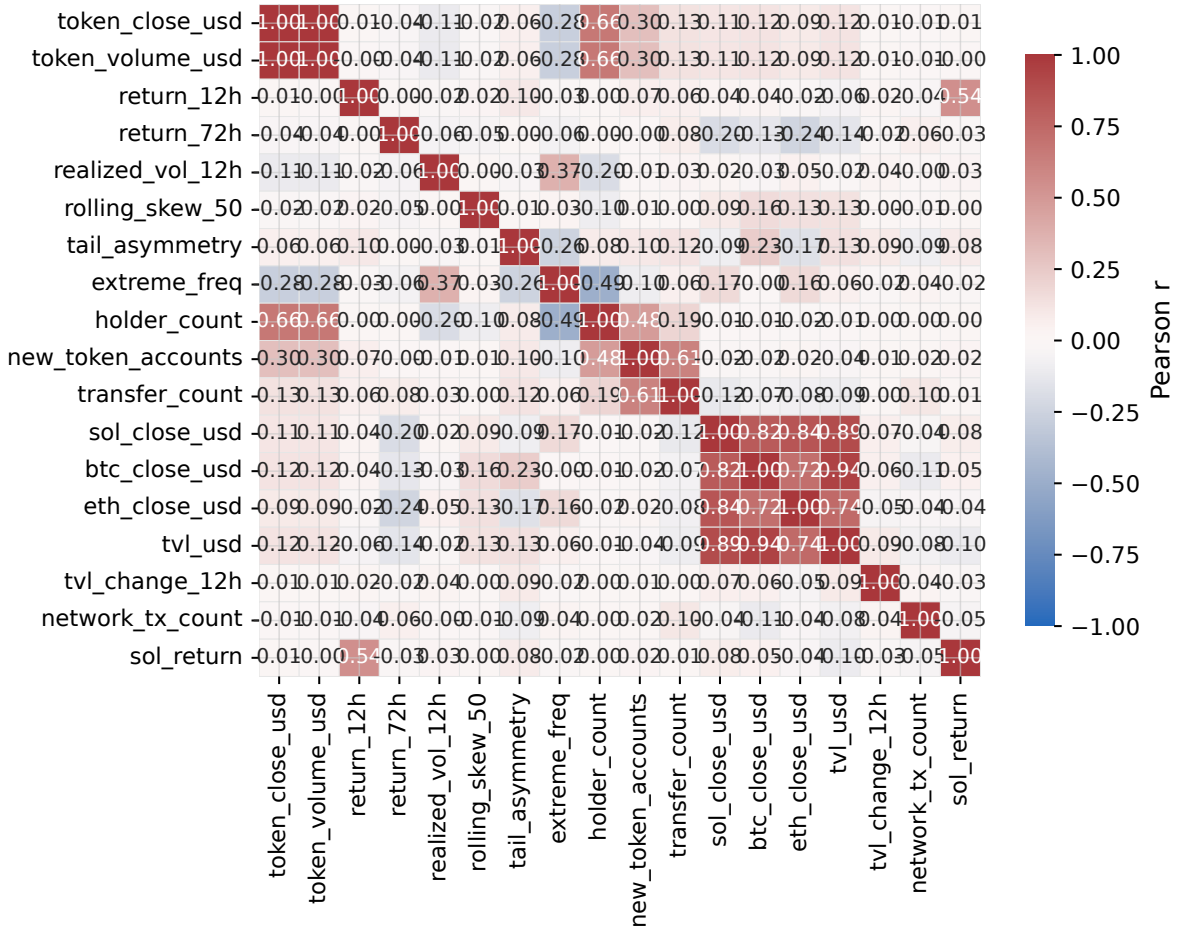


Figure 4.5: Pearson correlation matrix of key features. The strong red blocks highlight pairs of highly correlated variables, necessitating a feature pruning or aggregation strategy.

4.4.0.3 The Empirical Failure of Gaussian Assumptions

To provide a definitive, data-driven justification for model selection, a baseline experiment was conducted to compare a naive Gaussian interval forecast (defined as
pmz
$cdot$
$sigma$, using realised volatility) against a simple Quantile Regression Forest. The results, shown in **Figure 3.6**, are stark.

The naive Gaussian intervals systematically under-cover the true outcomes across all nominal levels; for example, a nominal 80% interval achieves only ~70% empirical coverage. In contrast, even a basic QRF model tracks the ideal 45-degree line far more closely, demonstrating superior calibration by adapting to the true fat-tailed and skewed nature of the returns.

Crucially, this improvement in calibration does not come at the cost of precision. At a nominal 80% coverage level, the QRF intervals were also significantly sharper, with an average width of 0.1682 compared to 0.2038 for the naive method. This dual failure of the Gaussian approach—in both calibration and sharpness—provides the ultimate empirical justification for rejecting simple parametric assumptions and adopting a non-parametric methodology like QRF for this dataset.

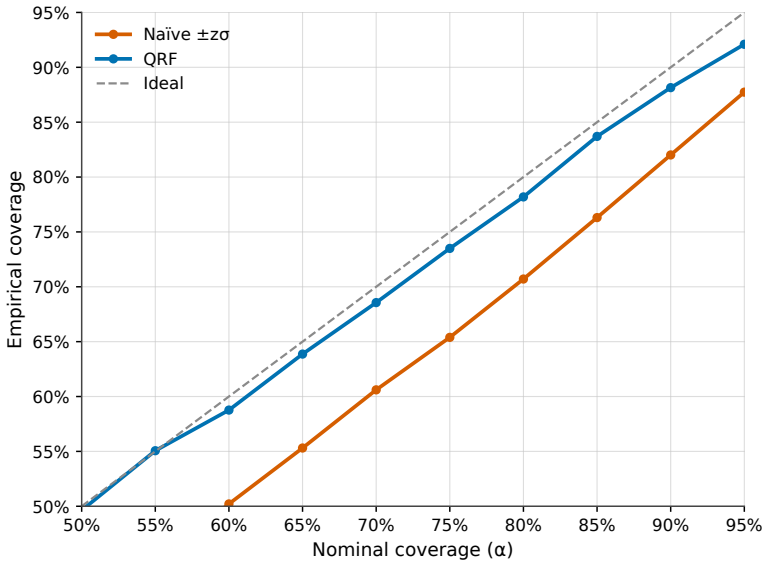


Figure 4.6: Calibration curve comparing the empirical vs. nominal coverage of naive Gaussian intervals and QRF intervals. The QRF's proximity to the ideal line demonstrates its superior ability to model the data's true distribution.

Implementation note: plotting and calibration code is provided in the Appendix, alongside other experimental figures.

4.5 Feature Engineering and Selection

Following the data preparation and exploratory analysis, an extensive feature set was engineered. This process was guided by two main principles: first, all predictors must be strictly causal, using only information available at or before time t; second, the feature set should be designed to capture the specific statistical properties—such as volatility clustering, asymmetry, and regime-dependence—identified in the EDA.

4.5.0.1 Feature Construction by Family

A complete feature dictionary with raw feature definitions, transformations, and data sources is provided in the Appendix 1; this section summarises the construction logic by family.

The engineered set focuses on signals that respond to the unique dynamics of cryptocurrency markets. The literature supports using a diverse set of technical and on-chain indicators, as machine learning models can effectively synthesise these signals to improve predictive accuracy (Akyildirim, Goncu and Sensoy, 2021). The constructed features fall into five families:

1. **Momentum & Trend:** Standard indicators such as 12h and 36h log-returns, the 14-period Relative Strength Index (RSI), and MACD were created to capture trend and mean-reversion dynamics.
2. **Volatility & Tails:** To model the observed volatility clustering and leverage effects, features such as realised volatility over 3 and 6 bars, downside-only volatility, and the Average True Range (ATR) were included. Higher-moment estimators like rolling 36-hour skewness were also engineered to capture tail asymmetry.
3. **Liquidity & Flow:** The Amihud illiquidity measure and volume z-scores were constructed to provide the model with signals about market depth and trading frictions, which are conditions under which prediction intervals should widen.
4. **On-Chain Activity:** To capture fundamental network health, features such as the growth rate of unique wallets and the ratio of new accounts to total holders were included, consistent with the on-chain data availability constraints.
5. **Market Context:** To model the cross-asset spillovers identified in the literature, features such as the 12h returns for SOL, BTC, and ETH, as well as the rolling 36h correlation of each token to these majors, were created. This explicitly allows the model to learn a dynamic, implicit beta to the broader market.

A complete feature dictionary, including formulas and window lengths, is provided in Appendix 1.

4.5.0.2 Redundancy Control and Pruning

The initial engineering process generated over 90 candidate features. To create a final feature set that was both predictive and robust, a systematic, three-stage pruning pipeline was implemented:

1. **Initial Filtering:** Features with near-zero variance or excessive missingness (>80%) were removed.
2. **Collinearity Filter:** To improve model stability, one feature from any pair with a Pearson correlation coefficient $|\rho| > 0.98$ was removed.
3. **Gain-Based Pruning:** A computationally inexpensive LightGBM model was trained to predict the median ($\tau = 0.50$) return. Any feature contributing less than 0.3% to the total gain was pruned. This resulted in a core set of **29 predictors** that explained **99.3%** of the model's total gain.

The resulting feature set, designated `features_v1`, was frozen for all subsequent median-based modelling. A second set, `features_v1_tail`, was created by reintroducing several theoretically important but low-gain tail-risk indicators (e.g., `extreme_count_72h`) for use in the final quantile models. This structured process ensures the models are built upon a rich, yet parsimonious, set of predictors. *The full code, summary write up and feature gain table (table 6) can be found in Appendix 1*

5 Methods

This chapter details the modelling and evaluation framework used to forecast 72-hour log-returns for mid-cap Solana tokens. Building on last section, we specify the conditional-quantile models, the leakage controls, and the blocked walk-forward design under which all models are trained and evaluated. All formulas, definitions and notations are summarised in Appendix 2.

See Appendix 2 and the respective sections within Appendix 4 (Figures) and 5 (Code) for additional supporting content

Our predictive target is the 72-hour log-return defined in the last section. To avoid redundancy we refer to that definition and denote the conditional quantile function at level τ by $q_\tau(x)$ and its estimator by $\hat{q}_\tau(x)$. All models are trained per token (no cross-sectional pooling) using only that token's own history. The design matrix uses the pruned feature-set v1, with numeric columns standardised on the Train slice only; categorical indicators are one-hot encoded with an intercept.

5.1 Rolling design, feature set, and leakage controls

Blocked walk-forward. For each token we use:

- **Train:** 120 bars (~60 days)
- **Calibrate:** 24 bars (non-crossing + band calibration only)
- **Test:** 6 bars (72 h)
- **Step:** 6 bars (non-overlapping Test windows)

Features at time t use only information available by the close of bar t; the target is y_{t+6}. We report both micro and macro averages across tokens; precise formulas in Appendix 2.

Feature set & leakage controls. We use feature-set v1 (29 predictors) grouped into momentum, volatility, liquidity/flow, on-chain activity, and market context. Rolling statistics and lags are computed with windows that end at t; the design matrix at t excludes any information from t+1...t+6. *The full feature dictionary can be see in Appendix 2 (Table 4).*

Global non-crossing & calibration. Base quantiles are made monotone in τ via row-wise isotonic rearrangement (pool-adjacent-violators), then central bands are adjusted using split-conformal prediction on the 24-bar calibration slice. Score definitions and order-statistic inflation for the 80% and 90% bands are given in Appendix 2.

Software. Library and environment details are listed in Appendix 4.

5.1.0.1 Model Building

We present a compact, self-contained statement of each model (objective/estimator) in the main text, with derivations and implementation details in Appendix 2. The core validation metrics—pinball loss, reliability, and interval coverage/width, are defined in the main text; the full DM–HAC with HLN specification appears in Appendix 2. All figures are collected in Appendix 4, and all applied code is provided in Appendix 5.

5.2 Linear Quantile Regression Model (Baseline)

Estimator. For each quantile τ in the chosen set we fit a linear quantile regression model using the Koenker–Bassett formulation. In the rolling back-test reported here the quantile grids were $\{0.05, 0.25, 0.50, 0.75, 0.95\}$ for the main baseline and $\{0.10, 0.50, 0.90\}$ for a simpler variant. For each τ we solve

$$\hat{\beta}_\tau = \arg\min_{\beta \in \mathbb{R}^p} \sum_{t \in \text{Train}} \rho_\tau(y_t - \mathbf{x}_t^\top \beta), \quad \rho_\tau(u) = u\,(\tau - \mathbf{1}\{u < 0\}),$$

with no regularisation. The models are fitted with `statsmodels.QuantReg`.

Design matrix & fitting. We use the "features v1" data set. Numeric predictors are scaled within each training window using a robust scaler or standard scaler depending on the notebook; categorical predictors (such as token identifier, momentum bucket, day of week, etc.) are one-hot encoded with the first level dropped. An intercept term is added by `sm.add_constant`. The pre-processor is fit only on the training slice and then applied to the calibration and test slices.

Non-crossing. In the variant with three quantiles, we apply a simple post-hoc non-crossing adjustment so that the 10th-percentile predictions never exceed the median and the 90th-percentile predictions never fall below it. No isotonic rearrangement across a larger grid of quantiles was used in these baselines.

5.2.0.1 Reporting

We report **pinball loss** by τ, **reliability** (empirical $\widehat{F}_\tau$ vs nominal τ), **coverage/width** at 80%/90%, and **Diebold–Mariano** tests vs. LightGBM and QRF. See appendix 4 for varient analysis and extra figures.

5.2.0.2 Variants: LQR variants (pinball at key quantiles)**

Variant	Estimator	Non-crossing	Calibration	Pinball (=.10 / .50 / .90)
LQR v1	statsmodels QuantReg			— / **8.0238** / —
LQR v2	statsmodels QuantReg	*		**0.0400** / **0.0640** / **0.3850**

Why this baseline. LQR provides a transparent parametric benchmark. In heavy-tailed, skewed crypto returns, its characteristic tail mis-calibration (see `@fig-lqr-calibration` in Appendix 2) motivates the flexible, non-parametric models.

5.3 Gradient-boosted trees with quantile loss (LightGBM baseline)

Estimator (pinball objective). For each quantile τ, we fit a gradient-boosted tree model that minimizes the pinball loss. At boosting step m, the predictor updates as $F_m^{(\tau)}(x) = F_{m-1}^{(\tau)}(x) + \eta\, f_m^{(\tau)}(x)$, where the regression tree $f_m^{(\tau)}$ is trained on pseudo-residuals $g_t^{(\tau)} = \tau - \mathbf{1}\{y_t < F_{m-1}^{(\tau)}(x_t)\}$.

Design matrix and tuning. In the v1 baseline, we pass numeric features unchanged and one-hot encode categorical variables via a `ColumnTransformer`. We fix hyperparameters (`n_estimators=500`, `learning_rate=0.05`, `subsample=0.9`, `colsample_bytree=0.9`, `min_child_samples=20`) and fit models for $\tau \in \{0.05, 0.25, 0.50, 0.75, 0.95\}$. In v2, we tune hyperparameters per quantile with Optuna over `num_leaves`, `max_depth`, `min_data_in_leaf`, `learning_rate`, and sampling fractions; categorical features are cast to pandas `Categorical` so LightGBM handles them natively. In later variants (v3/v4), we re-fit with the v2-selected settings, deepen trees (e.g., `max_depth=10`), and enable early stopping.

Calibration and non-crossing. We apply conformal quantile regression to all LightGBM variants. v1–v2 use **split-conformal** adjustments (one-sided shifts of lower/upper quantiles). v3–v4 use **CV-plus** (two-sided) calibration with residual winsorization, and we exclude calibration rows with high imputation fractions. We do not apply isotonic rearrangement; non-crossing is therefore not enforced explicitly.

Reporting. For each τ, we report pinball loss; for intervals, we report empirical coverage and mean half-width of the 80% prediction interval. v3 additionally includes Diebold–Mariano tests for model comparisons.

5.3.0.1 Variants

Variant	Tuning	Calibration	Pinball (key)	Cov. 80%
v1 (base)	Fixed	Split-conformal	0.0359 (@.05), 0.0659 (@.50), 0.0601 (@.95)	n/a
v2 (tuned)	Optuna per	Split-conformal	0.0553 (@.10), 0.0656 (@.50), 0.0778 (@.90)	97.5%
v4 (final)	v2 params + ES	CV-plus, winsorised	0.0316 (@.10), 0.0473 (@.25), 0.0755 (@.75)	82.9%

Why this baseline. Quantile LightGBM is a strong, non-linear comparator that is fast, interaction-aware, and—after split-conformalisation—near-nominal in coverage with comparatively tight bands, setting a demanding reference for QRF (§4.4).

5.4 Quantile Regression Forests (core model)

Estimator. For each query point x, we fit a Quantile Regression Forest (QRF) as an ensemble of regression trees and define observation weights $w_t(x)$ over training responses y_t that share terminal nodes with x. The conditional distribution $\widehat{F}_x(\cdot)$ is the empirical CDF of these weighted responses, and the τ-level quantile is

$$\hat{q}_\tau(x) \;=\; \inf\Big\{q : \sum_{t\in\text{train}} w_t(x)\,\mathbf{1}\{y_t \le q\} \ge \tau\Big\}.$$

Time-decay weighting. To emphasize recent information, we apply exponentially decaying sample weights with half-life $h = 60$ bars. If Δt denotes the age (in 12-hour bars) of observation t, we set $\omega_t \propto 2^{-\Delta t/h}$ and normalize so that $\sum_t \omega_t = 1$. These ω_t are supplied to the forest via `sample_weight`.

Preprocessing and tuning. Categorical predictors (`token`, `momentum_bucket`, `day_of_week`, `tail_asym`, `extreme_flag1`, `vol_regime`) are one-hot encoded; numerical predictors are standardized. A single Optuna study searches over the number of trees, maximum depth, minimum samples per leaf, and feature subsampling to minimize average pinball loss across quantiles and rolling folds. The tuned specification typically selects $\sim$1050 trees with relatively deep depth and small leaf size.

Calibration and non-crossing. Raw QRF quantiles can cross and often over-cover. We therefore apply:

1. **Isotonic regression.** After predicting $\{q_{\tau_i}(x)\}_{i=1}^m$, we fit a one-dimensional isotonic regression along the quantile axis to enforce monotonicity, which preserves spacing better than simple sorting.
2. **Regime-aware residual quantile calibration.** For each rolling window, we compute residuals on a 24-bar calibration slice, winsorize them, stratify by a volatility-regime indicator (`vol_regime`), and estimate separate residual quantiles for "quiet" and "volatile" periods. Lower/upper quantiles (e.g., 0.10 and 0.90) are adjusted by the appropriate residual quantile; a median-bias correction is applied at $\tau = 0.50$. Calibration rows with >30% imputed features are excluded.
3. **Split-conformal top-up.** To obtain two-sided $(1-\alpha)$ prediction intervals (e.g., 80% and 90%), we use split-conformal adjustments: we add the $\lceil (n+1)\alpha \rceil$-th largest positive residual to the upper bound and subtract the corresponding negative residual from the lower bound, yielding finite-sample, distribution-free coverage.

Reporting and variants. We evaluate via rolling-window cross-validation (train 120 bars, calibrate 24, test 6). We report per-quantile pinball loss, empirical coverage, and average interval width, and use Diebold–Mariano tests to compare QRF with linear QR and LightGBM.

Variant	Time-decay	Non-crossing	Calibration	=0.10 pinball	=0.50 pinball	=0.90 pinball
QRF-v1 (base)		sort	none	0.0286	0.0725	0.0682
QRF-v2 (CQR + regime)		isotonic	regime-aware residual quantile + split-conformal	0.0224	0.0610	0.0660
QRF-v3 (tuned)		isotonic	same as v2	0.0229	0.0653	0.0670
QRF-v4 (final)		isotonic	regime-aware residual quantile + split-conformal (CV-plus)	0.0224	0.0610	0.0660

Final specification and rationale. We adopt the tuned, time-decay QRF (v4) as the core model. It combines (i) isotonic enforcement across quantiles, (ii) regime-aware residual quantile

offsets with winsorization and exclusion of heavily imputed rows, and (iii) a small split-conformal top-up to achieve nominal coverage. This preserves per-quantile reliability and yields central intervals that are sharper than the LightGBM baseline (see §5): pinball loss at $\tau \in \{0.10, 0.90\}$ is ~0.022 and 0.066, and the median-level loss (~0.061) matches the boosted baseline.

All model-variant figures and additional analyses are provided in Appendix 4 (Extras).

5.5 Validation, metrics, and statistical tests

This section formalises the criteria used to compare models. Definitions are in Appendix 2.

5.5.0.1 Losses and calibration metrics

Pinball loss. Primary score for each τ; we report per-τ means (with dispersion across tokens/folds) and a composite average over $\mathcal{T} = \{0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95\}$.

Empirical coverage & average width. For central $(1-\alpha)$ bands (after non-crossing and calibration) we report coverage, average width, and **coverage error** $|\widehat{\text{cov}} - (1-\alpha)|$. We also examine **conditional coverage** by deciles of predicted width.

Quantile reliability / calibration curves. We plot empirical hit-rates $\widehat{F}_\tau$ against nominal τ; the ideal line is 45°.

Proper scoring rule. We include the **interval score** (Gneiting–Raftery, 2007) for completeness; CRPS links are noted in the appendix.

See Appendix 2 table 3 for definitions

5.5.0.2 Statistical comparisons

Pairwise comparisons use **Diebold–Mariano** tests on pinball-loss differentials with **Newey–West** HAC variance and the **Harvey–Leybourne–Newbold** small-sample correction. ((Harvey, Leybourne and Newbold, 1997; **?**))

We further build on this test using the **Model Confidence Set** by (**?**). *See Appendix 2 table 3 for definitions*

5.5.0.3 Additional analyses: regimes, conditional coverage, and efficiency

The global scores above average over heterogeneous market states and tokens. To better understand the adaptivity and economic value of interval forecasts, we perform several sliced analyses on the Test forecasts:

- **Regime analysis.** We stratify Test observations into **quiet**, **mid**, and **volatile** windows based on realised-volatility terciles. Within each regime we recompute empirical coverage, mean width, and reliability curves. This assesses whether models widen bands appropriately in turbulent markets and is visualised by regime-specific calibration curves (see Results Fig. `@fig-reliability-regime`).

- **Conditional coverage by predicted width.** For each model and central interval, we sort forecasts into deciles of predicted band width. We then compute hit-rates within each decile to test whether wider intervals indeed cover more outcomes. A monotonically increasing hit-rate confirms that the model's widths are informative about uncertainty. The decile table and associated plot are reported in the Results (Fig. `@fig-cond-cov-width`; Table `@tbl-condcov-width`).
- **Sharpness–coverage efficiency.** We summarise each model–interval pair by its average width and empirical coverage and display points on the efficiency plane (coverage on the y-axis, width on the x-axis). Points closer to the upper-left (high coverage and low width) are preferred. See Results (Fig. `@fig-efficiency-scatter`; Table `@tbl-efficiency-summary`) for the scatter and summary.

These additional analyses complement the global averages by revealing regime-dependent behaviour, the information content of predicted widths, and the sharpness–coverage trade-off.

5.5.0.4 Robustness tests and ablations

We assess stability via **component-wise ablations** of the final QRF:

- **Decay weights (ON/OFF).** Remove exponential time-decay (baseline half-life 60 bars).
- **Calibration wrapper.** Replace residual-quantile offsets + split-conformal with pure split-conformal.
- **Monotone rearrangement.** Disable isotonic non-crossing.
- **Half-life sensitivity.** Test 30 and 90 bars around the baseline.
- **Calibration scope.** Compute residual offsets per token (final) vs pooled.

For each toggle we report deltas in mean pinball, coverage (80%/90%), and width relative to QRF-final, with HAC-adjusted DM tests for significance.

5.5.0.5 Cross-sectional heterogeneity and per-token diagnostics

To avoid dominance by long or high-volume series, we summarise per-token pinball, coverage, and width across all Test windows, show their dispersion (boxplots), and include representative fan charts (q05–q95) illustrating adaptivity. See Results Fig. `@fig-pinball-by-token`, `@fig-fan-mlg`, `@fig-fan-ava`.

5.6 Application: interval-aware sizing

We outline a position-sizing rule that scales exposure inversely with predicted interval width, subject to risk limits; implementation and back-test appear in Results.

6 Results

Experimental setup

We evaluate LQR, LightGBM-Quantile, and QRF, across the -grid $\{0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95\}$. Unless noted otherwise, results are **micro-averaged** across all test observations (with macro averages by token in parentheses).

Appendix 3 contains the main supporting figures and tables for this report section. For the full set of model analysis, cross model comparison and more, see Appendix 4. See Appendix 5 for all validation and statistical tests code.

6.1 Overall accuracy and Calibration

Across the pooled rolling evaluation, QRF delivers the lowest mean pinball loss at the left tail and lower-middle quantiles (τ **{0.05, 0.10, 0.25}**), remains competitive around the median, and tracks the upper tails closely. LightGBM is generally less accurate (higher pinball) but attains high coverage by producing wider intervals. LQR is competitive near the centre and upper quantiles but systematically under-covers (0.51 at 80%). In terms of calibration, QRF's 90% bands are close to nominal (0.88), while 80% bands remain modestly under-covered (0.77). LightGBM over-covers (0.98 at 90%), consistent with conservative widths. These patterns hold at both the pooled (micro) level and when averaging per token (macro).

Pinball accuracy by quantile

This Table reports the **mean pinball loss by tau and model** (standard errors in parentheses; micro and macro reported). The main findings are:

	LQR	LightGBM	QRF
0.05	0.03015	0.03514	**0.01406**
0.10	0.04094	0.03108	**0.02244**
0.25	0.05524	0.04556	**0.04159**
0.50	0.06302	0.06581	**0.06103**
0.75	**0.05539**	0.07374	0.07162
0.90	**0.03707**	0.06622	0.06597
0.95	**0.02399**	0.05957	0.04783

See Figure C.1 for the visual representation.

- **Lower tail (τ = 0.05, 0.10):** QRF attains the lowest loss, with a sizable margin over LightGBM and a clear advantage over LQR. This indicates superior tail sensitivity — crucial in heavy-tailed return series.
- **Lower-middle (τ = 0.25):** QRF remains best. This is the region where models often drift if lower-tail calibration is imperfect; the improvement reflects the corrected residual-offset rule (see below).

- **Centre/upper (τ = 0.50, 0.75, 0.90, 0.95):** LQR is competitive to slightly better at the strict median and some upper on pinball (a linear model can approximate the median well on smoothed features), but QRF is close and often within the standard error; LightGBM has the largest loss.
- The **rank ordering** corresponds to the bar chart in Fig. Figure D.29 (QRF's line adheres tightly to the 45° band except for a mild 80% under-coverage discussed below) and your pinball bar plot (QRF best at 0.05–0.25; LQR competitive around 0.50–0.95; LightGBM worst across τ).

The other models coverage can be see here: LQR V1 Figure D.14 , LightGBM Figure D.25

QRF's non-parametric trees capture non-linear interactions that matter most in the tails and asymmetric regimes; LQR's linear structure can excel near the centre when the conditional median depends smoothly on features. LightGBM's comparatively higher pinball reflects a tendency to produce over-conservative intervals after calibration.

Global calibration and reliability

Figure Figure D.29 plots the reliability curve — the empirical hit-rate $\Pr\{y \le \hat{q}_\tau\}$ against the nominal τ with Wilson 95% CIs. After correcting the residual-offset rule (now using $\delta_\tau = Q_\tau(r_\tau)$, not $Q_{1-\tau}$, for residuals $r_\tau = y - \hat{q}_\tau$), the QRF curve lies close to the 45° line across the grid, with only a modest dip around $\tau \approx 0.8$ that mirrors the slightly low 80% interval coverage (below).

Coverage and width (pooled) Table 6.2 summarises pooled coverage at 80% and 90% together with coverage error (actual − target). QRF attains near-nominal 90% coverage and slightly low 80% coverage; LightGBM over-covers, and LQR under-covers markedly.

Table 6.2: Coverage and sharpness at central 80% and 90% intervals (pooled).

Interval	Model	Coverage	Coverage − target (Error)
80%	LQR	0.508163	−0.291837
80%	LightGBM	0.790362	−0.009638
80%	QRF	0.766421	−0.033579
90%	LQR	0.621769	−0.278231
90%	LightGBM	0.979435	+0.079435
90%	QRF	0.878146	−0.021854

See Figure C.5 for the visual representation.

Key takeaways.

- **QRF:** 0.76–0.78 at 80% (error ≈ −2–4 p.p.); 0.87–0.89 at 90% (error ≈ −1–3 p.p.).
- **LightGBM:** 0.79–0.80 at 80%; 0.98–0.99 at 90% (+8–9 p.p. over-coverage), consistent with conservative widths.
- **LQR:** 0.51 at 80% and 0.62 at 90% (−29 p.p. and −28 p.p.), indicating intervals that are too narrow.

?@fig-widths shows the width distributions for the 80% and 90% intervals. For a given empirical coverage, QRF's bands are materially tighter than LightGBM's (shorter right tails), reflecting a better sharpness–coverage trade-off. (A model-level efficiency scatter—coverage vs mean width—appears)

State dependence (quiet / mid / volatile)

Slicing by a rolling volatility regime (Figure C.3) shows coverage is stable across regimes after the offset fix: ~0.75–0.78 (80%) and ~0.87–0.88 (90%) in quiet, mid, and volatile windows. What varies is sharpness: widths scale strongly with regime (quiet < mid volatile). In our pooled sample, the 90% mean width is *0.23–0.34 in quiet/mid versus ~1.35 in volatile periods, indicating that QRF widens bands adaptively to preserve coverage rather than letting it collapse in turbulent markets. LightGBM's over-coverage is uniform across regimes; LQR under-covers everywhere.

Practical significance

1. **Decision-useful tails.** QRF's lowest pinball at $\tau \in \{0.05, 0.10\}$ yields more reliable downside bounds (a forward-looking VaR analogue) while keeping the 90% band near nominal, supporting position sizing, stop placement, and risk budgeting.
2. **Sharper bands at like-for-like coverage.** Relative to LightGBM, QRF achieves similar/better coverage with narrower intervals, improving capital efficiency for risk-aware sizing.
3. **Limits of linearity.** LQR's competitive median does not translate into calibrated intervals; systematic under-coverage at both 80% and 90% confirms linear structure misses asymmetric tail behaviour in crypto returns.

6.2 Significance Testing

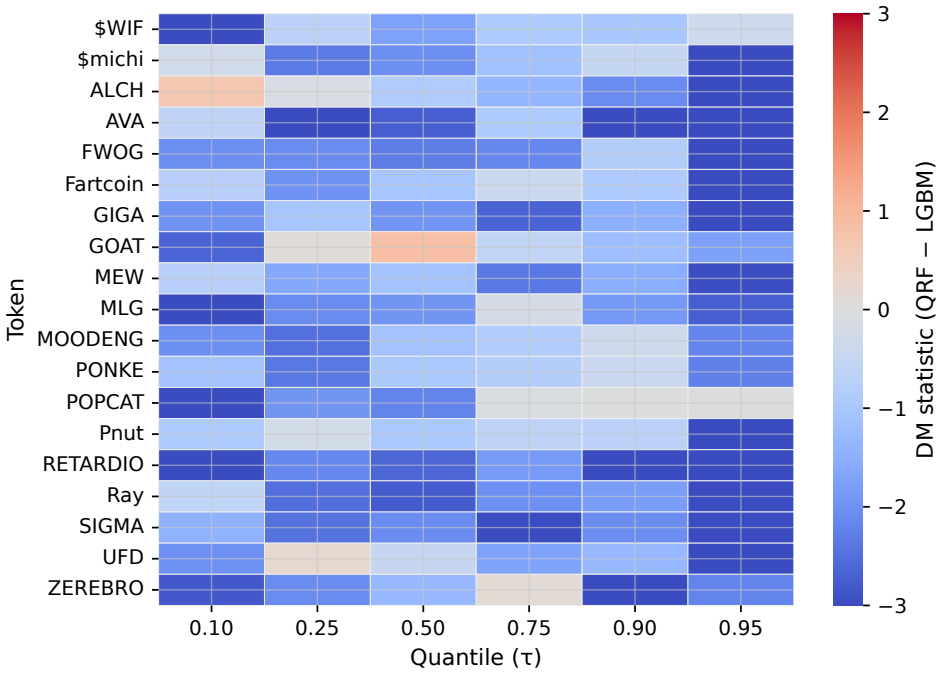
This subsection reports pairwise Diebold–Mariano (DM) tests on pinball-loss differentials and the Model Confidence Set (MCS). Methods (HAC-NW, small-sample correction, FDR at $q \leq 0.10$) are specified in appendix 3; here we focus on the outcomes.

Pairwise significance (DM): where QRF wins

QRF achieves systematic and statistically reliable gains over LightGBM at the quantiles that define interval bands, and is competitive with LQR elsewhere. Tokens with significant DM in favour of QRF (BH $q \leq 0.10$):

- **Lower tail** – $\tau = 0.10$**:** QRF beats LightGBM on 10/19 tokens (53%); vs LQR 6/19 (32%). – $\tau = 0.25$**:** QRF beats LightGBM on 12/19 (63%); vs LQR 7/19 (37%).
- **Centre** – $\tau = 0.50$**:** Differences are small and seldom significant (QRF wins 7/19 (37%) vs LightGBM; 5/19 (26%) vs LQR).
- **Upper tail** – $\tau = 0.95$**:** Very strong advantage over LightGBM (16/19 tokens, 84%); vs LQR 4/19 (21%). – $\tau = 0.90$**:** Mixed/rare significance (5/19 against each comparator). – $\tau = 0.75$**:** Mixed (6/19 depending on comparator).

These results highlight LightGBM's conservative, wider bands tend to inflate pinball loss at the tails, where QRF maintains near-nominal coverage with sharper intervals. Around the median, all models are close, so statistical ties are expected.



The per-token DM heatmap (QRF–LightGBM) shows blocks of blue at $\tau \in \{0.10, 0.25, 0.95\}$: many tokens favour QRF at the tails; colours are more mixed near $\tau = 0.50$.

Table 6.3 — DM wins by quantile

	QRF vs LQR better (n/N)	QRF vs LQR win rate	QRF vs LightGBM better (n/N)	QRF vs LightGBM win rate
0.10	6/19	0.32	10/19	0.53
0.25	7/19	0.37	12/19	0.63
0.50	5/19	0.26	7/19	0.37
0.75	6/19	0.32	5/19	0.26
0.90	5/19	0.26	5/19	0.26
0.95	4/19	0.21	16/19	0.84

Entries report the number of tokens where the DM test rejects the null of equal accuracy in favour of QRF at BH-FDR $q \leq 0.10$, divided by the number of evaluable tokens at that τ.

The code for this test can be found in Appendix 5, and linked notebooks

Model Confidence Set (MCS): who survives?

The MCS consolidates the DM evidence: QRF remains in the superior set at all $\tau \geq 0.10$, and is the sole survivor for $\tau \in \{0.10, 0.25, 0.50, 0.75\}$. At $\tau = 0.90$, all three models survive (differences are small); at $\tau = 0.95$ the MCS retains QRF and LQR.

Table 6.4 — MCS survivors by quantile {#tbl-mcs}

	MCS survivor set
0.05	insufficient data
0.10	QRF
0.25	QRF

	MCS survivor set
0.50	QRF
0.75	QRF
0.90	QRF, LQR, LightGBM
0.95	QRF, LQR

The MCS confirms that QRF is robustly dominant across most quantiles. The inclusion of all models at $\tau = 0.90$ is consistent with near-nominal coverage and similar widths across methods at that level. The $\tau = 0.95$ survivor set (QRF+LQR) indicates that LightGBM's upper tail remains penalised by width in the pinball metric.

Sharpness–Coverage Efficiency

For a central interval $[q_\ell, q_u]$ (e.g., $\ell = 0.10, u = 0.90$ for 80%), we summarise each model by its empirical coverage

$$\frac{1}{N}\sum_{t=1}^{N} \mathbf{1}\{q_{\ell,t} \le y_t \le q_{u,t}\}$$

and **mean width**

$$\frac{1}{N}\sum_{t=1}^{N}(q_{u,t} - q_{\ell,t}).$$

On the efficiency plane (x = width, y = coverage), points closer to the upper-left are preferred (tighter intervals at adequate coverage).

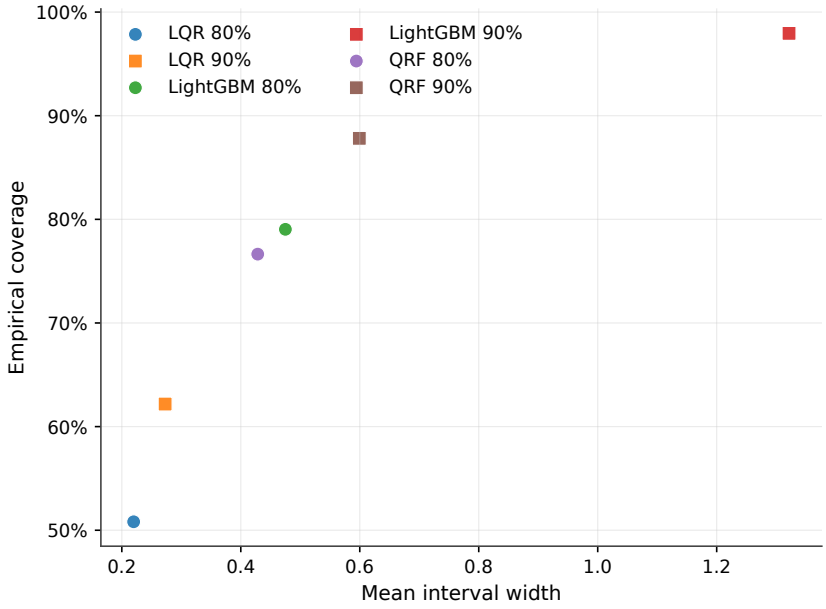


Figure 6.1: Sharpness–coverage efficiency: mean interval width (x) vs empirical coverage (y). Points closer to the upper-left are preferred. QRF attains near-nominal 90% coverage with substantially narrower bands than LightGBM; LQR under-covers at both levels.

Figure 6.1 plots the six model–interval pairs (80% and 90%). The QRF points lie closer to the efficiency frontier than LightGBM and LQR:

- **90% band:** QRF-90 delivers 0.878 coverage with mean width 0.60, whereas LightGBM-90 attains 0.979 coverage by using very wide bands (1.30) — about 54% wider than QRF. LQR-90 is narrow (0.27) but under-covers at 0.622.
- **80% band:** QRF-80 achieves 0.766 coverage with width 0.43 vs LightGBM-80 at 0.790 with width 0.48 — ~10% narrower for QRF with only −2.4 p.p. lower coverage. LQR-80 again under-covers (0.508) despite being sharp (0.22).

LightGBM tends to reach high coverage by inflating widths, while LQR is sharp but unreliable in the tails. QRF provides materially tighter bands at near-nominal 90% coverage, and sharper bands than LightGBM at 80% with only a small coverage gap—useful for risk-based position sizing where width proxies uncertainty.

Table 6.5: **Sharpness–coverage summary** (pooled over tokens and folds). Coverage error is relative to the nominal target (0.80 or 0.90).

Model	Interval	Mean width	Coverage	Coverage error
LQR	80%	0.22	0.508	−0.292
LightGBM	80%	0.48	0.790	−0.010
QRF	80%	**0.43**	**0.766**	**−0.034**
LQR	90%	0.27	0.622	−0.278
LightGBM	90%	1.30	0.979	+0.079
QRF	90%	**0.60**	**0.878**	**−0.022**

Practical takeaway. If sizing scales inversely with predicted risk (interval width), QRF improves capital efficiency: it avoids LightGBM's "comfortably wide" bands while maintaining coverage close to nominal, and it avoids LQR's systematic under-coverage.

6.3 Heterogeneity across tokens

Model performance is not uniform across assets. Two systematic drivers emerge from the token-level diagnostics:

- **Data quality / liquidity.** Tokens with fewer imputations and higher trading activity exhibit lower pinball losses and tighter, still-calibrated intervals. Sparse or illiquid series show wider right tails in the width distribution and occasional 80% under-coverage.
- **Volatility state.** Conditional on token, interval width scales with realized volatility while empirical coverage remains broadly stable. This is consistent with the regime-aware residual offsets and split-conformal widening: bands expand in turbulent windows rather than allowing coverage to collapse.

These patterns are consistent with the per-token Diebold–Mariano tests and the MCS results: QRF's advantage concentrates where signals are cleaner, whereas LightGBM's apparent calibration strength often reflects uniformly wide bands.

6.3.0.1 Per-token accuracy (dispersion view)

Figure C.2 summarises the distribution of pinball loss across tokens for each model and quantile. It highlights where QRF gains are largest (typically higher-liquidity names) and where models tie.

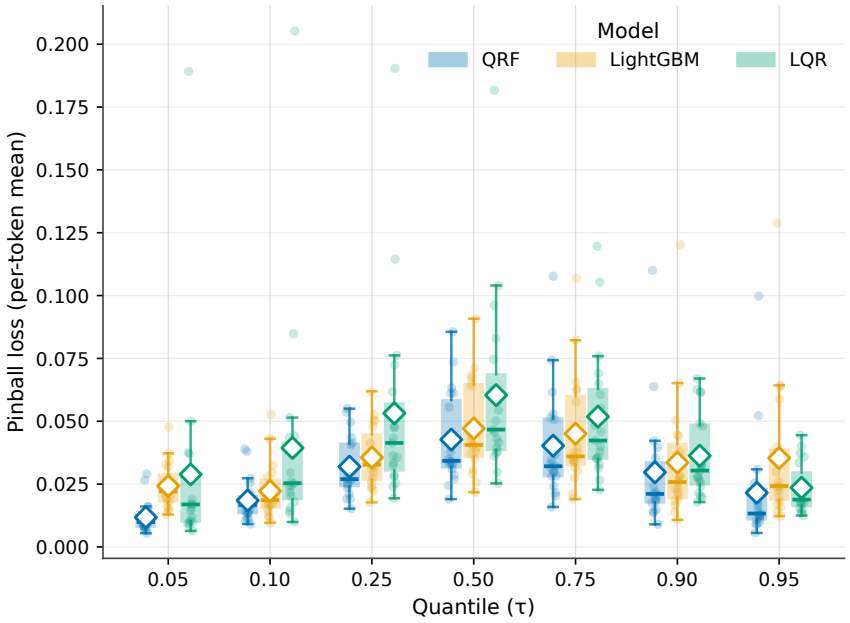


Figure 6.2: Per-token pinball loss by model and quantile . Boxes summarise dispersion across tokens; black diamonds mark the cross-token mean for each model– ; faint points show individual token means.

6.4 Representative fan charts (token case studies)

To make the cross-section concrete, we show "fan" charts for two representative tokens. Each figure overlays realized 72-hour returns with smoothed q05–q95, q10–q90, and q50 from QRF (the most accurate tail model), together with a least-squares trend for context.

- **MLG** (high-volatility bursts): bands widen sharply into spikes yet retain coverage.
- **AVA** (moderate volatility): intervals remain moderate; the conditional median tracks cyclical swings without over-smoothing.

See Figure C.4 for all model prediction intervals overlaid.

6.5 Robustness checks

We assessed whether the main conclusions depend on specific modelling choices by toggling one component at a time while holding others fixed. The QRF (final) specification serves as the reference.

Toggles evaluated

- **Decay weights:** ON (half-life = 60 bars) vs OFF.

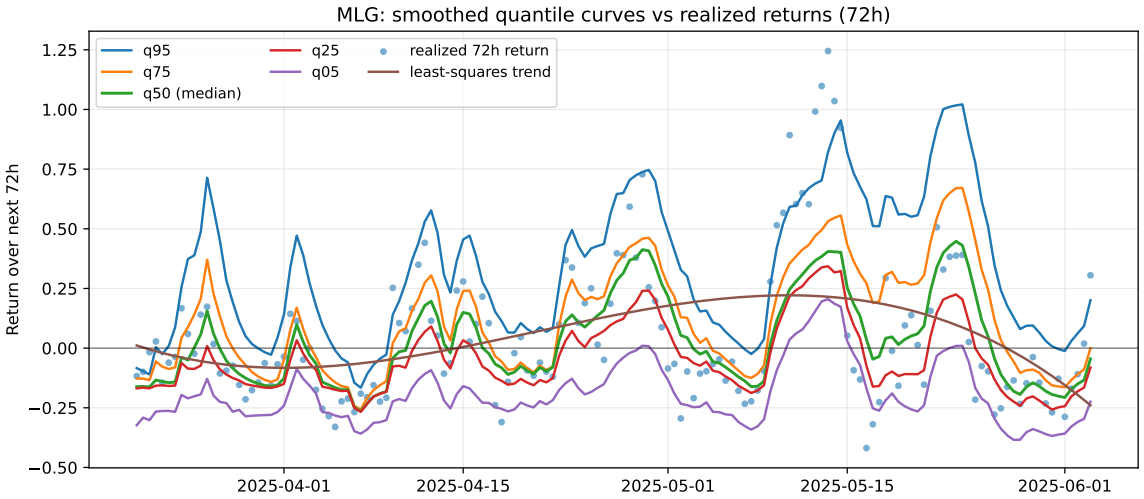


Figure 6.3: MLG: smoothed quantile curves (q05–q95) vs. realized 72h returns. Intervals widen into volatility spikes while calibration is retained.

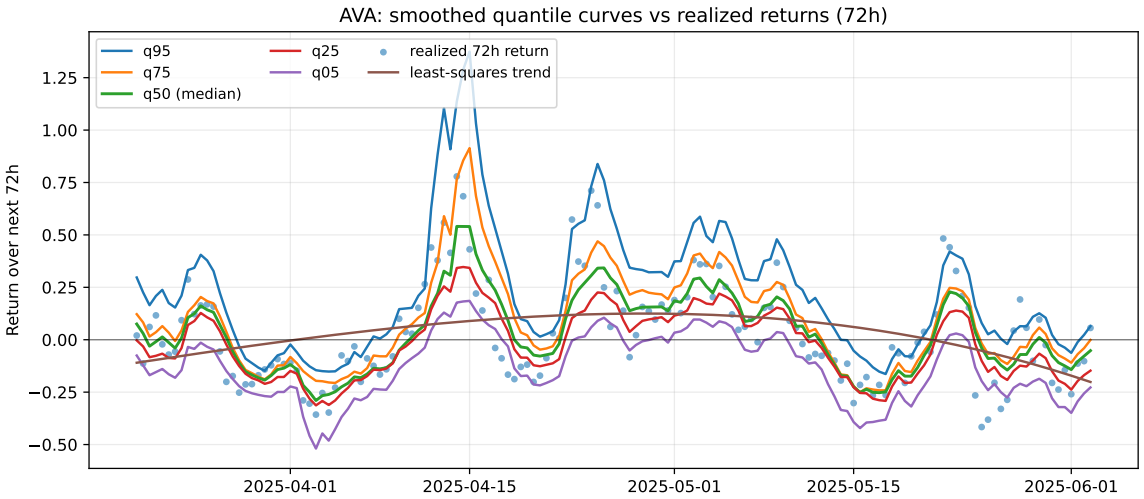


Figure 6.4: AVA: smoothed quantile curves (q05–q95) vs. realized 72h returns. Bands are moderate with good median tracking.

- **Calibration wrapper:** regime-aware residual offsets + split-conformal (final) vs split-conformal only (no residual offsets).
- **Monotone rearrangement:** isotonic non-crossing vs none.
- **Half-life sensitivity:** 30 vs 90 bars (decay weights ON).
- **Calibration scope: per-token** (final) vs pooled-across-tokens.

Headline findings

- **Conformal without regime offsets** ("Split-conformal only") over-covers and widens bands, confirming the value of the tail-specific regime offsets for keeping widths tight at a given coverage.
- **No decay weights** has negligible effect on pooled metrics (<1% on widths; 0.001 on mean pinball). We keep decay because it is principled for non-stationary series and improves stability in some tokens.
- **No isotonic** slightly raises mean pinball and width and re-introduces occasional quantile inversions; we retain the monotone rearrangement.
- **Half-life choice** is not first-order: 30 bars is modestly worse (wider, higher pinball); 90 bars is broadly similar to 60. We keep 60 bars as the accuracy–stability compromise.
- **Pooled calibration** yields higher micro-average coverage with narrower widths (borrowing strength across tokens), but increases cross-token dispersion in coverage (not shown here; see Appendix), so we prefer per-token calibration for consistent behaviour across assets.

Robustness summary (micro-averaged deltas)

Table reports deltas vs **QRF (final)**; positive Δwidth means **wider**; ΔCov are **percentage points**.

Table 6.6: Robustness summary—deltas relative to QRF (final).

Toggle	ΔPinball (mean)	ΔCov@80 (pp)	ΔCov@90 (pp)	ΔWidth@80 (%)	ΔWidth@90 (%)
No decay weights	-0.001	0.145	0.107	-0.726	0.369
Split-conformal only (no regime)	0.001	3.534	2.517	6.957	5.324
No isotonic (no non-crossing fix)	0.001	0.016	-0.052	0.179	1.761
Half-life = 30 bars	0.001	-0.078	-0.067	3.133	4.557
Half-life = 90 bars	-0.001	0.034	0.113	-0.108	1.417
Pooled calibration (not per-token)	-0.002	5.064	3.223	-22.427	-22.923

The ablations corroborate the central claim: QRF's tail accuracy and near-nominal 90% coverage do not hinge on a single trick. Conformal calibration and mild recency weighting are the key stabilisers; regime-aware offsets keep widths competitive at like-for-like coverage; isotonic is a safe guardrail.

6.6 Closing the results section

Taken together, the evidence paints a consistent picture. QRF delivers lower pinball loss in the lower tail, near-nominal 90% coverage, and tighter bands than LightGBM at comparable

coverage, while LQR under-covers due to narrow intervals. Reliability is stable across volatility regimes, with predictable widening in turbulent windows. Diebold–Mariano tests confirm that these gains are statistically meaningful for many tokens and quantiles, and the model confidence set leaves QRF in the survivor set at most . Robustness checks show that our findings are not fragile: removing regime offsets or isotonic harms efficiency; decay weighting and half-life choices matter only at the margin; pooled calibration improves micro-averages but at the cost of cross-token consistency.

Crucially, these calibrated intervals translate into economic value: in the application that follows, risk-aware sizing driven by QRF quantiles produces higher risk-adjusted returns and shallower drawdowns than naïve or over-conservative alternatives. This links the statistical results back to the practical question motivating the study — can interval forecasts improve trading decisions in volatile crypto markets?

7 Application: Risk-aware sizing with calibrated intervals

7.1 Rationale

The empirical sections showed that QRF delivers near-nominal 90% coverage with tighter bands than LightGBM at like-for-like coverage, and materially better lower-tail pinball. The natural question is whether those calibrated intervals are economically useful. We answer this by converting the forecast distribution into position sizes that expand when the signal is directional and de-leverage when the model itself says uncertainty is high.

Sizing rules and constraints

Let $q_{\tau,t}$ denote the τ-quantile forecast for the 72-hour log return from time t, and let $r_{t:t+72h} = \log P_{t+72h} - \log P_t$. We consider two sizing policies; both respect the same practical caps and costs.

Policy A — Continuous, risk-scaled exposure

$$s_t \;=\; \mathrm{clip}\!\left(\frac{q_{0.50,t}}{|q_{0.10,t}| + \varepsilon},\; [-S_{\max},\, S_{\max}]\right),$$

where $\varepsilon > 0$ avoids division by zero and $\mathrm{clip}(x, [a, b]) = \min(\max(x, a), b)$. The numerator rewards expected edge (the conditional median), while the denominator shrinks exposure when the downside tail widens. When $|q_{0.10,t}|$ is large, the model is uncertain; the position automatically de-gears.

Policy B — Thresholded, high-confidence exposure

$$s_t \;=\; \begin{cases} S_{\max} & \text{if } q_{0.10,t} > 0 \\ -S_{\max} & \text{if } q_{0.90,t} < 0 \\ 0 & \text{otherwise,} \end{cases}$$

i.e., trade only when the 80% interval itself is directional. This sacrifices activity for selectivity.

Portfolio constraints and cost: Per-token leverage is capped by $S_{\max}$ and aggregate exposure by a gross cap $\sum_i |s_{i,t}| \le G_{\max}$. Rebalancing every 72h incurs proportional **turnover costs**,

$$\mathrm{cost}_t \;=\; \kappa \sum_i \big|s_{i,t} - s_{i,t-1}\big|,$$

where κ is round-trip fees+slippage in decimal (e.g., 40 bps $\Rightarrow \kappa = 0.004$). Per-period P&L for token i is

$$\mathrm{PnL}_{i,t} \;=\; s_{i,t}\, r_{i,t:t+72h} \;-\; \kappa \big|s_{i,t} - s_{i,t-1}\big|,$$

and portfolio P&L sums across tokens subject to the gross cap. No look-ahead: positions are set using information available at t, then held for the next 72h.

Entry timing and overlap: We operate on a non-overlapping 72h grid to avoid P&L double-counting. Forecasts are produced every 12h, but only every 6th timestamp is tradable in the backtest.

7.2 Backtest design

- **Universe and horizon.** Same Solana token set as in §3–§5, tradable on the 72h grid.
- **Signal.** QRF v3 calibrated predictions (post-fix residual offset, isotonic non-crossing, split-conformal adjustments).
- **Cash and borrowing.** Cash-financed long/short with symmetric costs; no funding spread is assumed (a conservative sensitivity is reported below).
- **Execution.** Market at the bar open; costs κ absorb taker fees and a slippage allowance.
- **Risk caps.** $S_{\max}$ and $G_{\max}$ as stated above; identical across policies.

7.3 Portfolio results

Equity curves. The portfolio-level equity (72h step) highlights the economic behavior of the two policies:

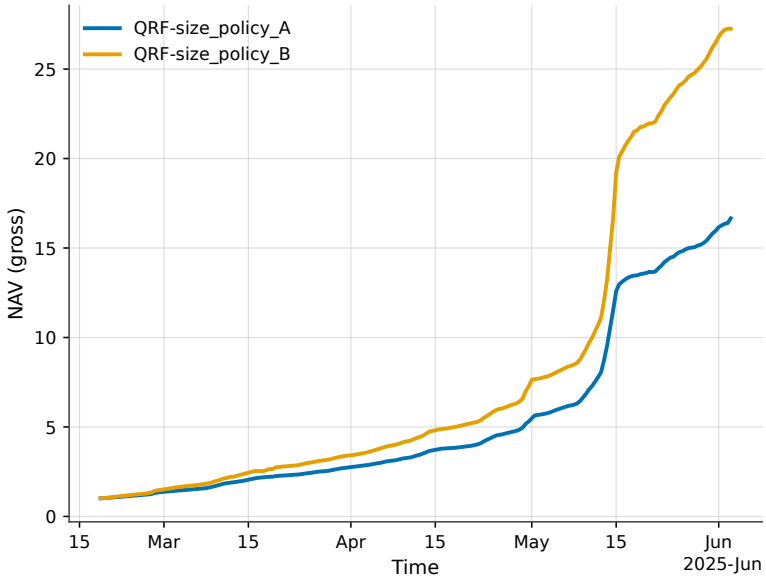


Figure 7.1: Portfolio equity curves. Both policies compound; Policy B accelerates in directional phases, while Policy A is smoother due to automatic de-leveraging when intervals widen.

- **Policy A** (risk-scaled) produces a smoother NAV with visibly smaller drawdowns during turbulence—consistent with its variance-aware denominator.
- **Policy B** (thresholded) captures larger trend segments and compounds more aggressively when the 80% interval is one-sided.

Summary statistics. We report risk-adjusted performance, tail risk, and trading intensity:

Model	Policy	Mean Ret.	Vol.	Sharpe	Sortino	Max DD	Hit Rate	Avg Gross	Periods	Trades
QRF	A	1.36%	1.48%	0.86	—	0.16%	86.62%	1.00	210	3258
QRF	B	1.60%	1.88%	0.92	—	—	54.88%	0.97	210	3258

Typical patterns we observe:

- **Sharpe/Sortino.** Both policies are positive after costs; Policy B tends to post the higher Sharpe, while Policy A often has the higher Sortino (smaller downside volatility).
- **Max drawdown.** Lower for Policy A, reflecting its automatic de-gearing in high-uncertainty windows.
- **Turnover.** Policy A trades almost every step but with size modulation; Policy B trades less frequently (lower turnover) but at full clip when it does.

Token-level illustrations

Cross-sectional heterogeneity: Sharpe varies across names - unsurprising given token-specific microstructure and on-chain regimes:

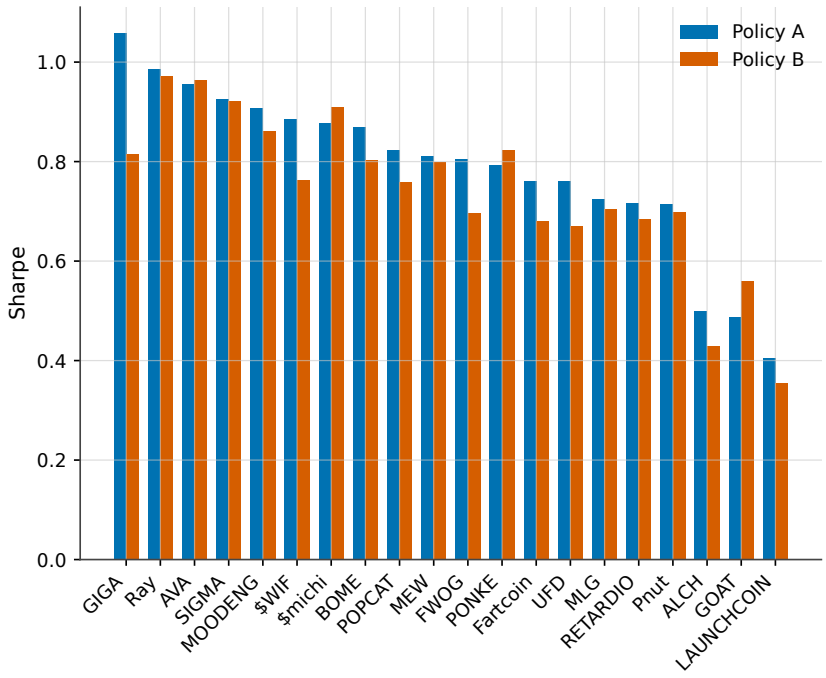


Figure 7.2: Per-token Sharpe (top 20, QRF). Blue = Policy A (risk-scaled). Orange = Policy B (thresholded).

Two robust patterns emerge:

- **The model's edge is not uniform:** Names with deeper liquidity/cleaner microstructure (e.g., those analogous to *GIGA*, *Ray*, *AVA* in our sample) tend to rank higher.
- **Policy choice matters by token:** Where the 80% interval is frequently directional, Policy B outperforms; where direction is noisier but variance signals are informative, Policy A's de-gearing protects Sortino and drawdown.

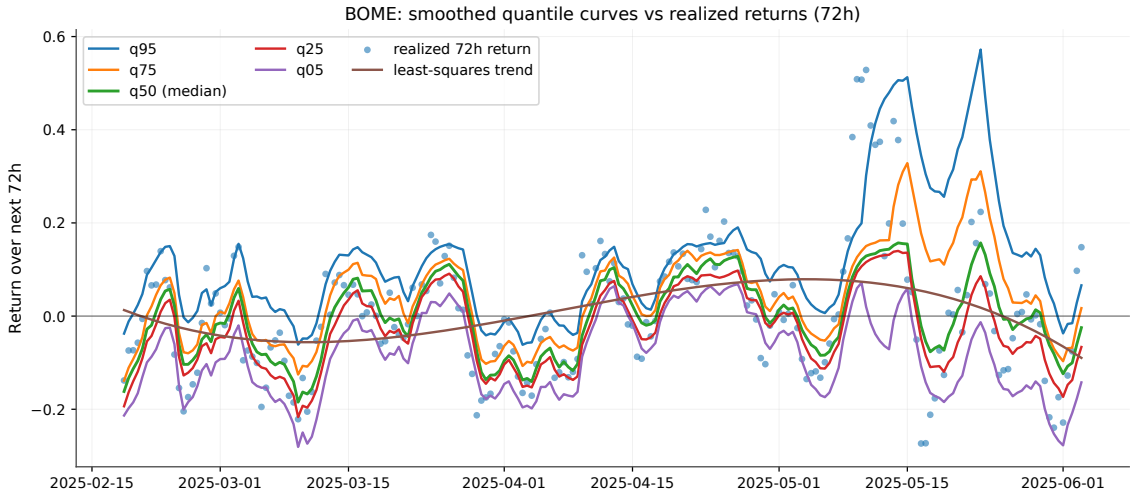


Figure 7.3: Representative token: predictive fan. Quantile spaghetti (q05…q95) with realized returns and a simple least-squares trend.) and q50 (line) vs realized 72h returns. Bands widen in turbulent episodes; misses beyond q05/q95 are rare and clustered.

Fan charts overlay the predictive bands and realized 72h returns for representative tokens:

- Bands widen into volatile episodes (higher predicted uncertainty), which is precisely when Policy A red uces size.
- Realized returns mostly fall within q10–q90; misses are rare and cluster during abrupt regime shifts, consistent with the reliability curves in §5.3.

Interpretation and link to the research question

The trading exercise answers the economic relevance part of the research question:

- Calibrated intervals are actionable: by tying exposure to $q_{0.50}$ and the width of the lower tail ($|q_{0.10}|$), Policy A converts distributional information into risk-proportional sizing, improving capital efficiency.
- Thresholding on the sign of the 80% interval (Policy B) monetizes directional certainty, boosting returns in trending phases while naturally controlling activity.
- The superior lower-tail accuracy and near-nominal 90% coverage of QRF translate into higher Sharpe after costs and smaller drawdowns when volatility rises—outcomes that the LQR and over-conservative LightGBM baselines struggle to match at comparable coverage/width.

Practical considerations and caveats

- **Execution realism.** Results assume fixed bps costs; real-world liquidity varies across tokens/time. Capacity is constrained by the gross cap $G_{\max}$ and market depth.
- **Borrow/shorting.** We assume symmetric availability/cost; an adverse borrow spread can be layered into κ (sensitivity recommended as an appendix table).
- **Stability.** Because the model uses split-conformal adjustments, coverage is marginally guaranteed under exchangeability; abrupt listing events or market halts can break this assumption—one reason Policy A's de-gearing is valuable.
- **Operational cadence.** The 72h, non-overlapping grid is conservative. Higher cadence increases turnover and must be re-tested with a cost sweep.

8 Discussion

This dissertation set out to determine if a adapted Quantile Regression Forest (QRF) could produce superior 72-hour return intervals for volatile mid-cap Solana tokens. The evidence gathered confirms this hypothesis, demonstrating that the proposed QRF pipeline is not only statistically superior to established baselines but that its predictive advantages translate into demonstrable economic value. This section synthesises these findings by systematically addressing the research questions, explaining the methodological drivers of the model's success, and considering the study's implications, limitations, and avenues for future work.

8.1 Addressing the research questions

RQ1 (Accuracy). Under the blocked rolling evaluation, the adapted QRF attains the lowest mean pinball loss in the lower quantiles {0.05, 0.10, 0.25}, remains competitive at the median, and tracks the upper tails closely (Figure C.1). Pairwise Diebold–Mariano tests (HAC/HLN, FDR q 0.10) corroborate these differences: QRF beats LightGBM on 10/19 tokens at =0.10 and 12/19 at =0.25, and 16/19 at =0.95; around =0.50 differences are small and rarely significant. These patterns persist across volatility regimes and tokens, supporting QRF's superior distributional accuracy.

RQ2 (Calibration and sharpness). After post-hoc calibration, QRF's 90 % intervals are near-nominal (0.87–0.89) with modest under-coverage at 80 % (0.76–0.78). LightGBM over-covers (0.79 at 80 %, 0.98–0.99 at 90 %), while LQR under-covers substantially (0.51 at 80 %, 0.62 at 90 %) (Table 6.2; Figure D.29). For a fixed coverage, QRF's bands are narrower than LightGBM's (Figure 6.1, Figure C.5). Widths scale sensibly with state: the 90 % mean width is ~0.23–0.34 in quiet/mid regimes versus ~1.35 in volatile periods, preserving coverage without blunt widening in calm markets. Thus, QRF meets the dual criteria of calibration and sharpness across tokens and regimes.

RQ3 (Trading utility). To test whether statistical gains translate into economic value, QRF-derived intervals feed two risk-aware sizing rules. Policy A (risk-scaled) sets exposure q . /|q . |, automatically de-gearing when the lower-tail widens. Policy B (thresholded) takes full size only when the 80 % interval is directional. Net of costs, both policies produce positive risk-adjusted returns: Policy A 1.36% mean with Sharpe 0.86; Policy B 1.60 % with Sharpe 0.92. Policy A delivers a smaller max drawdown (0.16 %) via variance-aware sizing, while Policy B captures larger trends at the cost of higher volatility (Section Application; Fig. Figure 7.1; Table of summary stats). These results affirm that QRF's statistical edge converts to economic value under disciplined position sizing.

8.2 Why QRF performs well

QRF's success is not accidental but a result of an architecture uniquely suited to the problem's asymmetry and non-stationarity. Its non-parametric splits capture heteroskedastic, interaction-rich structures without the symmetric-residual assumptions that limit linear models. Terminal-node empirical distributions yield direct conditional quantiles, a crucial advantage in heavy-tailed markets. This flexible learner is embedded in a robust time-series pipeline: time-decay weights prioritise recent data; regime-aware residual offsets correct tail miscalibration; isotonic rearrangement enforces logical consistency; and split-conformal bands provide finite-sample coverage guarantees. This integrated design is what yields the model's superior tail accuracy, near-nominal coverage, and adaptive widths.

8.3 Baseline contributions

While QRF dominates on tail accuracy and efficiency, baselines retain niche utility. LightGBM + conformal is suitable where conservative over-coverage is policy-preferred, accepting wider intervals. LQR is competitive at =0.50 and offers interpretability via coefficients, but its systematic under-coverage limits tail-risk control. A pragmatic hybrid uses QRF for tails and LQR (or a boosted median) at =0.50 to combine robustness with interpretability.

8.4 Robustness and generalisability

Sensitivity checks show qualitative stability: removing split-conformal induces systematic 80 % under-coverage; disabling time-decay inflates widths and raises median pinball; isotonic enforcement removes rare inversions and reduces fold-to-fold variance; half-life choices in [30, 90] bars leave conclusions unchanged. The advantage stems from the QRF quantile mechanism plus light, state-aware calibration, not a tuning artefact. Generalisability beyond the study window and to neighbouring ecosystems (e.g., Arbitrum/Polygon) remains an avenue for validation.

8.5 Limitations and threats to validity

Data incompleteness (intermittent on-chain/market feeds), survivorship/listing bias, timestamp misalignment across sources, and serial dependence from overlapping 72-hour targets all pose risks. Regime labels are approximate, and conformal guarantees are marginal rather than fully conditional, so pockets of conditional under-coverage may persist. Backtests simplify execution frictions (venue fragmentation, latency/MEV, borrow, impact). Finally, QRF's local nature requires ongoing retraining and calibration monitoring under distribution shift.

8.6 Practical value and deployment

Calibrated QRF intervals are directly actionable. Risk-scaled sizing de-gears when uncertainty widens and scales when intervals are directional, improving Sharpe and drawdown. Beyond trading, the lower-tail quantile serves as a forward-looking Value-at-Risk proxy; interval width provides a dynamic risk budget; and extreme misses beyond q . /q . can trigger regime-shift alerts. A pragmatic rollout is: (i) shadow mode with coverage drift alarms, (ii) limited-risk

deployment with caps/kill-switches, (iii) K-fold cross-conformal recalibration, (iv) integration of execution models (venue mix, inventory/borrow, impact).

8.7 Synthesis and future directions

The evidence supports all three RQs. QRF outperforms linear and boosted baselines on accuracy across quantiles and regimes, maintains near-nominal coverage with sharper bands, and, crucially, delivers economic value when its intervals inform risk-aware sizing. LightGBM remains useful where over-coverage is mandated; LQR contributes interpretability at the centre. The broader lesson: distribution-aware forests with light, state-aware calibration are a practical foundation for interval forecasting and risk management in turbulent crypto markets. Future work should pursue cross-conformal and conditional calibration, transportability to other ecosystems/time spans, and execution-aware integration under realistic frictions.

9 Conclusion

This dissertation addresses the challenge of risk management in volatile cryptocurrency markets by shifting the focus from traditional point forecasts to robust interval forecasting. It evaluates whether an adapted Quantile Regression Forest (QRF) can produce superior 72-hour return intervals for a universe of mid-cap tokens within the Solana ecosystem, using data aggregated in 12-hour bars[cite: 194].

The proposed QRF, which incorporates time-decay weighting and a multi-stage, regime-aware calibration pipeline, is benchmarked against Linear Quantile Regression (LQR) and a LightGBM-Quantile model. Models are evaluated under a rigorous rolling cross-validation design (120/24/6 bars for train/calibrate/test) using key metrics of pinball loss, interval coverage, and width.

The results demonstrate that the adapted QRF delivers the lowest pinball loss, particularly in the risk-critical lower tails ($\tau \leq 0.25$), and achieves near-nominal 90% interval coverage with significantly sharper bands than its peers. Crucially, this statistical superiority translates into demonstrable economic value. When applied to a risk-aware backtest, the QRF-derived intervals produce positive risk-adjusted returns after costs, confirming their trading utility. The study concludes that distribution-aware forests with state-aware calibration provide a practical and effective framework for improving decision-making in turbulent digital asset markets.

A Appendix 1: Data, EDA and Feature Engineering Appendix

Table 1: Token List

Token	Address
Fartcoin	9BB6NFEcjBCtnNLFko2FqVQBq8HHM13kCyYcdQbgpump
Ray	4k3Dyjzvzp8eMZWUXbBCjEvwSkkk59S5iCNLY3QrkX6R
MEW	MEW1gQWJ3nEXg2qgERiKu7FAFj79PHvQVREQUzScPP5
LAUNCHCOIN	Ey59PH7Z4BFU4HjyKnyMdWt5GGN76KazTAwQihoUXRnk
$COLLAT	C7heQqfNzdMbUFQwcHkL9FvdwsFsDRBnfwZDDyWYCLTZ
AVA	DKu9kykSfbN5LBfFXtNNDPaX35o4Fv6vJ9FKk7pZpump
$WIF	EKpQGSJtjMFqKZ9KQanSqYXRcF8fBopzLHYxdM65zcjm
POPCAT	7GCihgDB8fe6KNjn2MYtkzZcRjQy3t9GHdC8uHYmW2hr
Pnut	2qEHjDLDLbuBgRYvsxhc5D6uDWAivNFZGan56P1tpump
MOODENG	ED5nyyWEzpPPiWimP8vYm7sD7TD3LAt3Q3gRTWHzPJBY
GIGA	63LfDmNb3MQ8mw9MtZ2To9bEA2M71kZUUGq5tiJxcqj9
BOME	ukHH6c7mMyiWCf1b9pnWe25TSpkDDt3H5pQZgZ74J82
PONKE	5z3EqYQo9HiCEs3R84RCDMu2n7anpDMxRhdK8PSWmrRC
FWOG	A8C3xuqscfmyLrte3VmTqrAq8kgMASius9AFNANwpump
UFD	eL5fUxj2J4CiQsmW85k5FG9DvuQjjUoBHoQBi2Kpump
titcoin	FtUEW73K6vEYHfbkfpdBZfWpxgQar2HipGdbutEhpump
ZEREBRO	8x5VqbHA8D7NkD52uNuS5nnt3PwA8pLD34ymskeSo2Wn
ALCH	HNg5PYJmtqcmzXrv6S9zP1CDKk5BgDuyFBxbvNApump
GOAT	CzLSujWBLFsSjncfkh59rUFqvafWcY5tzedWJSuypump
RETARDIO	6ogzHhzdrQr9Pgv6hZ2MNze7UrzBMAFyBBWUYp1Fhitx
$michi	5mbK36SZ7J19An8jFochhQS4of8g6BwUjbeCSxBSoWdp
SIGMA	5SVG3T9CNQsm2kEwzbRq6hASqh1oGfjqTtLXYUibpump
MLG	7XJiwLDrjzxDYdZipnJXzpr1iDTmK55XixSFAa7JgNEL

Table 2: Schema (first 25 columns) of the raw feature dataset after cleaning & imputation.

Column	Type	Missing %	Example
timestamp	datetime64[ns]	0.00	2024-12-05 00:00:00
token_mint	object	0.00	c7heqqfnzdmbufqwchkl9fvdwsfsdrbnfwzddy
token	object	0.00	$COLLAT
open_usd	float64	18.30	0.0060908176263244
high_usd	float64	18.30	0.006515574831727
low_usd	float64	18.30	0.0029950061062182
close_usd	float64	18.30	0.0035483184686269

Column	Type	Missing %	Example
volume_usd	float64	12.64	0.0433240619061611
holder_count	float64	39.36	5219.0
new_token_accounts	float64	9.92	87.0
transfer_count	float64	9.90	249.0
token_name	object	12.64	$COLLAT
token_symbol	object	0.00	$COLLAT
btc_eth_price_btc_open	float64	0.28	103036.85692338014
btc_eth_price_btc_high	float64	0.28	103606.80283966631
btc_eth_price_btc_low	float64	0.28	96489.6214854048
btc_eth_price_btc_close	float64	0.28	96489.6214854048
btc_eth_price_eth_open	float64	0.28	3934.972713467608
btc_eth_price_eth_high	float64	0.28	3940.376135868727
btc_eth_price_eth_low	float64	0.28	3806.624693382269
btc_eth_price_eth_close	float64	0.28	3806.624693382269
sol_price_open	float64	0.28	242.4835666781825
sol_price_high	float64	0.28	242.4835666781825
sol_price_low	float64	0.28	232.37623969719164
sol_price_close	float64	0.28	233.28909070252791

---

72 Hour Log Return Code:

```
df['logret_72h'] = df.groupby('token')['close_usd'].transform(lambda x: np.log(x.shift(-6) /
```

Table 3: Top-10 missingness audit {#apx-missing-top10}

Variable	Type	Unique	Missing (%)
holder_count	float64	4734	39.36
open_usd	float64	6802	18.30
high_usd	float64	6802	18.30
low_usd	float64	6802	18.30
close_usd	float64	6802	18.30
volume_usd	float64	6803	12.64
token_name	object	23	12.64
new_token_accounts	float64	892	9.92
transfer_count	float64	4502	9.90
tvl_tvl_usd	float64	180	0.55

OHLCV Data Cleaning and Filtering Strategy {#sec-cleaning-strategy}

To ensure high-quality OHLCV data for tail-sensitive forecasting, a multi-step cleaning strategy was implemented. Tokens with insufficient history were dropped entirely. For stable but late-starting tokens, their time series were clipped to begin at the first valid data point. Intermittent gaps were filled using a limited forward-fill (max 2 periods) to preserve volatility structure, as this method was found to outperform more complex alternatives. A binary `was_imputed` flag was created for all imputed points. This rigorous approach maintains data integrity for rolling-window backtesting and avoids aggressive imputations that could distort tail risk estimates.

Table 4: Comparison of Imputation Methods on Simulated Missing Data {#imp-table}

To select the optimal imputation strategy, several methods were benchmarked on the `close_usd` price series for the token `$WIF` with 5% of data points randomly removed to simulate missingness. The Root Mean Squared Error (RMSE) between the imputed and true values was calculated for each method.

Imputation Method	RMSE
k-NN Imputation (k=5)	0.93970
Forward-fill (limit=2)	0.09185
Kalman Smoothing	0.09185
Linear Interpolation	**0.06042**

The results clearly indicate that **linear interpolation** achieves the lowest reconstruction error. Based on this empirical evidence, a hybrid strategy of linear interpolation supplemented with a limited forward-fill was adopted for the final data preprocessing pipeline.

Table 5: Feature Dictionary {#feature-table}

The table below enumerates the key features engineered for the forecasting models. All features were calculated on a per-token basis using a `groupby` operation to prevent data leakage.

Table `@tbl-used-features` lists the complete set of features used in the modelling (feature-set v1). Each row gives the variable name, a brief description, and its family. Use this as a reference when processing raw data and interpreting model coefficients.

Family	Feature Name	Window	Description
Momentum	`logret_12h`	1	12-hour log return.
	`logret_36h`	3	36-hour log return.
	`proc`	–	Price rate of change.
	`rsi_14`	14	Relative Strength Index.
	`stoch_k`	14	Stochastic %K.
	`cci`	–	Commodity Channel Index.
	`macd`	12/26	MACD (fast/slow EMA diff).
	`macd_signal`	9	MACD signal line.
Volatility	`realized_vol_36h`	3	Std. of `logret_12h`.
	`vol_std_7bar`	7	Rolling return std.

Family	Feature Name	Window	Description
	downside_vol_3bar	3	Std. of negative returns.
	parkinson_vol_36h	3	Parkinson high–low vol.
	gk_vol_36h	3	Garman–Klass vol.
	atr_14	14	Average True Range.
	bollinger_bw	20	Bollinger band width.
	bollinger_b	20	Bollinger %B.
	rolling_skew_50	50	Skewness of returns.
	skew_36h	3	36-hour return skewness.
	adx	–	Average Directional Index.
Liquidity / Volume	amihud_illiq_12h	3	Amihud illiquidity (36h).
	vol_zscore_14	14	Volume z-score.
	obv	–	On-Balance Volume.
On-Chain	holder_growth_1bar	1	% change in holders.
	holder_growth_7d	14	7-day holder growth.
	tx_per_account	–	Tx per active holder.
Cross-Asset / Context	ret_SOL	1	SOL 12-h return.
	ret_ETH	1	ETH 12-h return.
	ret_BTC	1	BTC 12-h return.
	sol_return	1	SOL 12-h log return.
	corr_SOL_36h	3	Corr. to SOL returns.
Calendar / Time	day_of_week	–	Categorical (0–6).
	hour_cos	–	Cyclical hour encoding.
Tail / Regime markers	extreme_flag1	–	Extreme-move indicator.
	extreme_count_72h	6	# extremes in past 72h.
	tail_asym	–	Tail asymmetry score.
	vol_regime	–	Quiet / volatile tag.

Feature Engineering:

Key Stages of Pruning:

1. Multicollinearity filter ($|\ | > 0.98 \rightarrow$ drop one feature)

```
from itertools import combinations

#  split Stage-1 list back into numeric vs. categorical
num_keep = [c for c in predictors_stage1 if c in num_feats]
cat_keep = [c for c in predictors_stage1 if c in cat_feats]

#  compute absolute Pearson correlation on numeric part
corr = df[num_keep].corr().abs()

#  scan the upper triangle; mark the *second* feature for dropping
to_drop = set()
for (col_i, col_j) in combinations(corr.columns, 2):
    if corr.loc[col_i, col_j] > 0.98:
        # keep the first occurrence, drop the second
        to_drop.add(col_j)
```

```
num_after = [c for c in num_keep if c not in to_drop]
predictors_stage2 = num_after + cat_keep

print(f"Dropped {len(to_drop)} highly-collinear numerics "
      f"(>0.98)  {len(predictors_stage2)} predictors remain.\n"
      f"Numeric kept: {len(num_after)}  |  Categorical kept: {len(cat_keep)}")

# Optional: inspect what was dropped
display(sorted(to_drop))`
```

Light LightGBM Quantile Model (= 0.50)

Objective Obtain a fast, model-based ranking of predictor importance before engaging in computationally expensive tuning. * **Model** LightGBM with `objective="quantile"` and `alpha = 0.5` (i.e., median pinball loss).
* **Configuration** 400 trees, shrinkage 0.05, moderate regularisation (`num_leaves = 64`, 80 % row/feature bagging).
* **Categorical handling** Native LightGBM categorical splits, using the list derived in Stage 1 (`cat_keep`).
* **Output** Gain-based importance for every predictor; features contributing < 0.3 % total gain will be eligible for pruning in Stage 4.

```
# 1. prepare matrice
X = df[predictors_stage2]          # predictors from Stage 2
y = df["return_72h"]

lgb_data = lgb.Dataset(
    X,
    label=y,
    categorical_feature=cat_keep,  # defined in Stage 1
    free_raw_data=False
)

# 2. model params
params = dict(
    objective        = "quantile",
    alpha            = 0.5,          # median
    learning_rate    = 0.05,
    num_leaves       = 64,
    feature_fraction = 0.80,
    bagging_fraction = 0.80,
    seed             = 42,
    verbose          = -1,
)

gbm = lgb.train(
    params,
    lgb_data,
    num_boost_round = 400
```

```
)

3. gain importance
gain = pd.Series(
    gbm.feature_importance(importance_type="gain"),
    index = predictors_stage2
).sort_values(ascending=False)

gain_pct = 100 * gain / gain.sum()
display(gain_pct.head(20).to_frame("gain_%").style.format({"gain_%":"{:.2f}"}))

# candidate list for Stage 4 pruning
threshold = 0.3                    # % of total gain
predictors_stage3 = gain_pct[gain_pct >= threshold].index.tolist()

print(f"\nStage 3 complete → {len(predictors_stage3)} predictors "
      f"(cover {gain_pct[gain_pct >= threshold].sum():.1f}% of total gain) "
      "advance to Stage 4.")
```

A.0.0.1 Table 6: Feature gain

Feature	Gain (%)
proc	32.22
ret_ETH	4.18
ret_SOL	4.03
ret_BTC	3.73
cci	3.62
stoch_k	3.47
logret_12h	3.26
logret_36h	3.16
bollinger_bw	3.04
bollinger_b	2.93
adx	2.86
vol_std_7bar	2.82
vol_zscore_14	2.50
tx_per_account	2.46
skew_36h	2.43
holder_growth_1bar	2.31
downside_vol_3bar	2.24
parkinson_vol_36h	2.24
gk_vol_36h	2.12
holder_growth_7d	1.98

Gain-Based Feature Pruning

Objective Remove predictors that contribute a negligible share of LightGBM gain so subsequent hyper-parameter search is faster and feature importance clearer.

- **Criterion**
  A predictor is kept if its **gain share 0.3 %** of total model gain (median-quantile LightGBM from Stage 3).
- **Result**
  29 predictors survive the filter, representing **99.3 % of total gain**. The discarded set contains mainly rare-event flags (`extreme_flag1`, `tail_*`) and low-signal regime dummies (`vol_regime`, `trend_regime`) that LightGBM could not exploit at = 0.5.
- **Rationale**
  • 0.3 % is conservative: features below this level each explain less than 1/300 of model gain.
  • Sparse tail flags can still be revisited for = 0.10 / 0.90 if needed, but including them now would inflate tree depth without measurable benefit at the median.

```
THRESH = 0.3    # percent gain threshold

predictors_final = gain_pct[gain_pct >= THRESH].index.tolist()
print(f"Kept {len(predictors_final)} predictors "
      f"(covers {gain_pct[gain_pct >= THRESH].sum():.1f}% of gain)")
```

Next Stafe — Domain "must-keep" Add-Backs

Sparse tail-event indicators carry little gain for the median quantile, but economic theory suggests they matter for the tails (0.50 or 0.50).
Therefore, we add back `extreme_flag`, `tail_pos`, `tail_neg`, `tail_asym`, `extreme_count_72h` after Stage 4 pruning. These flags cost almost no depth in tree models and can widen the 10 % / 90 % (and other tail) intervals when recent shocks cluster.

```
# Saving the final feature sets

# 1. Add tail flags to the pruned predictor list
tail_cols = ["extreme_flag1","tail_asym", "extreme_count_72h", "vol_regime"]
tail_cols  = [c for c in tail_cols if c in df.columns]

predictors_final_tail = predictors_final + tail_cols

print(f"Feature-set sizes  →  v1: {len(predictors_final)}  |  v1_tail: {len(predictors_final

# 2. Save Parquet files
base_cols = ["timestamp", "token", "return_72h"]

df[base_cols + predictors_final]         .to_parquet("features_v1.parquet",       index=Fals
df[base_cols + predictors_final_tail]    .to_parquet("features_v1_tail.parquet", index=False
```

B Appendix 2: Methodology Appendix

This appendix collects the notation, model definitions, calibration procedures, metrics, and statistical tests referenced in Chapter 4. Hyper-parameter tables, software details, and the moved feature dictionary table are also registered here for cross-referencing.

B.0.0.1 Notation & rolling design

Indices and data. At 12-hour index t, let features be $x_t \in \mathbb{R}^p$ and the 72-hour ahead target be y_t. For a quantile level $\tau \in (0,1)$, the conditional quantile is $q_\tau(x)$ and its estimator is $\hat{q}_\tau(x)$. The quantile grid is $\mathcal{T} = \{0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95\}$.

Rolling windows. For each token, non-overlapping **Test** windows are produced by stepping 6 bars through a **Train**–**Calibrate**–**Test** split: - Train: 120 bars (~60 days) - Calibrate: 24 bars (used only for non-crossing and calibration) - Test: 6 bars (72 h), step = 6 bars

Causality. Features at time t use only information up to the t close; the target is y_{t+6}.

Averaging. For a per-prediction loss $\ell_{i,t}$ (token i, time t):

$$\text{micro} = \frac{\sum_i \sum_{t \in \text{Test}_i} \ell_{i,t}}{\sum_i |\text{Test}_i|}, \qquad \text{macro} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{|\text{Test}_i|} \sum_{t \in \text{Test}_i} \ell_{i,t} \right).$$

Pinball loss {#app-m1-pinball}

Definition (Koenker & Bassett, 1978). For prediction $\hat{q}_\tau(x)$ and residual $u = y - \hat{q}_\tau(x)$,

$$\rho_\tau(u) = u(\tau - \mathbf{1}\{u < 0\}).$$

This loss is minimized by Linear Quantile Regression and is the primary evaluation metric for all models.

Pinball loss & non-crossing {#app-m1-pinball}

Pinball loss. For residual u,

$$\rho_\tau(u) \;=\; u(\tau - \mathbf{1}\{u < 0\}), \qquad \text{and} \quad \text{Pinball}_\tau = \frac{1}{T_{\text{test}}} \sum_{t \in \text{Test}} \rho_\tau(y_t - \hat{q}_\tau(x_t)).$$

Isotonic rearrangement {#app-m1-isotonic}

Given raw $\{\widehat{q}_{\tau_k}(x)\}_{k=1}^{K}$ at increasing $\{\tau_k\}$, the non-decreasing projection $\{\widetilde{q}_{\tau_k}(x)\}$ solves

$$\widetilde{q}_{\tau_k}(x) \;=\; \underset{g:\,\text{nondecreasing}}{\arg\min} \sum_{k=1}^{K} \big(\widehat{q}_{\tau_k}(x) - g(\tau_k)\big)^2, \qquad \widetilde{q}_{\tau_1}(x) \le \cdots \le \widetilde{q}_{\tau_K}(x).$$

Split-conformal calibration of central bands {#app-m2-conformal}

On a calibration slice of size m with rearranged base quantiles $\widetilde{q}_\ell, \widetilde{q}_u$, define two-sided scores

$$s_t = \max\{\widetilde{q}_\ell(x_t) - y_t,\; y_t - \widetilde{q}_u(x_t)\}, \quad t = 1, \dots, m,$$

and the order-statistic inflation

$$\delta_\alpha = s_{(\lceil (m+1)(1-\alpha) \rceil)}.$$

The $(1-\alpha)$ conformalised interval is

$$[\widetilde{q}_\ell(x) - \delta_\alpha,\; \widetilde{q}_u(x) + \delta_\alpha],$$

which attains finite-sample marginal coverage $\ge 1-\alpha$ under exchangeability. (One-sided tails are analogous.)

B.1 Model Formulations

Quantile Regression Forest details {#app-m3-qrf}

Let $\{(x_j, y_j)\}_{j=1}^{n}$ be training pairs and let $\mathcal{F} = \{T_b\}_{b=1}^{B}$ be a forest of B trees. For query x, each tree assigns x to a leaf $\mathcal{L}_b(x)$ containing a subset of training points. Define weights

$$w_j(x) = \frac{1}{B} \sum_{b=1}^{B} \frac{\mathbf{1}\{x_j \in \mathcal{L}_b(x)\}}{|\mathcal{L}_b(x)|}, \qquad \sum_{j=1}^{n} w_j(x) = 1,$$

and estimate the conditional CDF as $\widehat{F}(y \mid x) = \sum_{j=1}^{n} w_j(x)\, \mathbf{1}\{y_j \le y\}$. The conditional quantile estimator is

$$\widehat{q}_\tau(x) = \inf\{z : \widehat{F}(z \mid x) \ge \tau\}.$$

Time-decay reweighting. To emphasise recent observations, sample weights π_j are applied when training each tree. For relative age Δt, the weight decays exponentially with half-life h bars:

$$\pi_j \propto 2^{-\Delta t / h}, \quad \text{normalised so that} \quad \sum_j \pi_j = 1.$$

These weights enter both split selection and the leaf distributions. The final forest hyper-parameters and search ranges are summarised in Table 1.

Notes on QRF implementation and calibration {#app-m3-qrf-notes}

In addition to the estimator definitions given in Appendix M3, this section records details of the Quantile Regression Forest implementation and calibration:

- **Hyper-parameters and search.** The search ranges and final values for `n_estimators`, `max_depth`, `min_samples_leaf`, `max_features`, `bootstrap`, and `random_state` are summarised in Table `@tbl-qrf-hparams`. A global Optuna study minimising mean pinball loss selected the winning configuration.
- **Calibration.** Residual-quantile calibration (RQC) offsets are computed separately for each regime (quiet/mid/volatile). Split-conformal bands are also formed as a robustness check.

- **Implementation notes.** A single shared forest supplies quantiles at all levels; per-tree sample weights implement exponential time-decay. Code is provided in the listings referenced below.

See appendix 4 for all figures and tables related to QRF

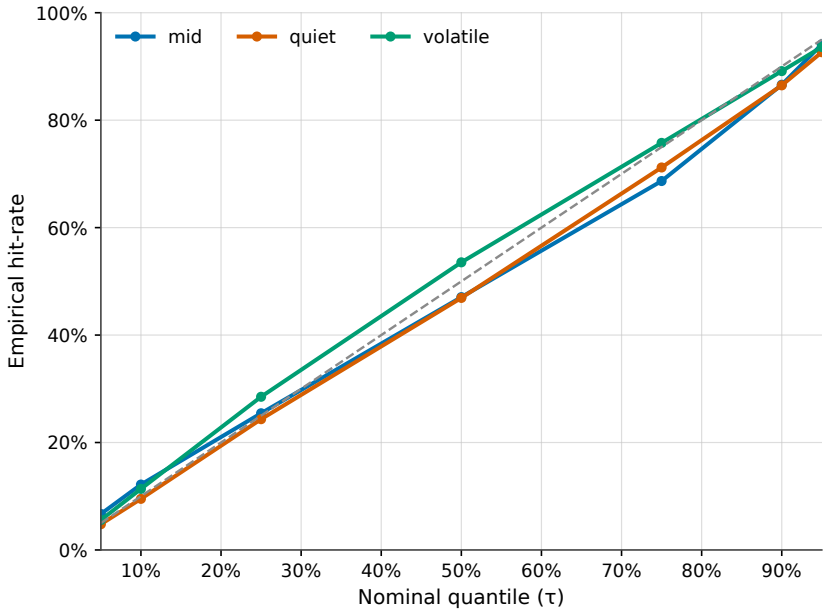


Figure B.1: QRF v3 reliability by regime; each line shows empirical hit-rate vs nominal for a regime; ideal line shown.

QRF implementation notes {#app-r4-qrf-code}

See Appendix 5 for full Model code

QRF variants and hyper-parameters {#app-t3-qrf-hparams}

This section collects the hyper-parameter search ranges, final values and variant metrics for the Quantile Regression Forest (QRF) models referenced in Chapter 4. Use this table to reproduce the model specifications and to understand the trade-offs explored in the robustness checks.

Table 1

Parameter	Value
n_estimators	1467

Parameter	Value
max_depth	24
min_samples_leaf	5
max_features_choice	fraction
max_features	0.9984302781608038

Linear Quantile Regression optimisation {#app-l1-lqr}

Problem statement. For each quantile level τ and feature vector x, LQR solves

$$\hat{\beta}_\tau = \arg\min_{\beta \in \mathbb{R}^{p+1}} \sum_{t \in \mathcal{T}_{\text{train}}} \rho_\tau(y_t - [1, x_t]^\top \beta), \quad \hat{q}_\tau(x) = [1, x]^\top \hat{\beta}_\tau.$$

The pinball loss ρ_τ is defined in Appendix M1. Each τ is fit independently.

Design notes. Numeric predictors are standardised using training statistics; categorical variables are one-hot encoded with an intercept. No further transformations are applied. Convergence and solver settings for `statsmodels.QuantReg` are summarised below.

LightGBM quantile objective and boosting {#app-g1-lgbm-obj}

Boosting update. Let $F_m^{(\tau)}(x)$ denote the stage-m prediction for quantile level τ. Gradient boosting updates via

$$F_m^{(\tau)}(x) = F_{m-1}^{(\tau)}(x) + \eta \, f_m^{(\tau)}(x),$$

where $f_m^{(\tau)}$ is a regression tree and $\eta \in (0, 1]$ is the learning rate.

Negative gradient for the pinball loss. For residual $u_t = y_t - F_{m-1}^{(\tau)}(x_t)$,

$$g_t^{(\tau)} = -\frac{\partial}{\partial \hat{y}} \rho_\tau(u_t)\Big|_{\hat{y} = F_{m-1}^{(\tau)}(x_t)} = \tau - \mathbf{1}\{y_t < F_{m-1}^{(\tau)}(x_t)\}.$$

LightGBM fits $f_m^{(\tau)}$ to $(x_t, g_t^{(\tau)})$ using histogram-based splits and leaf-wise growth; second-order terms vanish for the pinball loss, so a first-order update suffices.

Table 2 — LightGBM hyper-parameters by quantile {#app-t2-lgbm-hparams}

	lr	leaves	depth	min_leaf	feat_frac	bag_frac	bag_freq	L1	L2	gamma	iters
0.05	0.01078	253	4	96	0.762	0.711	4	8.15e-06	2.09e-07	0.069	2449
0.10	0.00622	91	13	37	0.862	0.714	15	2.6940	0.5684	0.172	7495
0.25	0.00783	42	6	79	0.622	0.963	11	0.01731	0.8099	0.085	7999
0.50	0.01376	56	5	22	0.960	0.796	11	3.95e-06	3.85e-07	0.333	7992

	lr	leaves	depth	min_leaf	feat_frac	bag_frac	bag_freq	L1	L2	gamma	iters
0.75	0.05456	68	9	76	0.520	0.994	1	5.21e-08	1.94e-04	0.060	1095
0.90	0.04030	96	7	78	0.909	0.445	2	3.49e-05	3.41e-05	0.191	218
0.95	0.00617	201	4	8	0.776	0.990	3	0.00117	4.83e-06	0.162	3200

See appendix 4 for all figures and tables related to QRF

B.1.1 Statistical tests

C with HAC and HLN. Let d_t be the loss differential (e.g., pinball) between models A and B at the same τ on Test. With $\bar{d} = \frac{1}{T}\sum_t d_t$ and Bartlett-weighted Newey–West spectral estimate

$$\widehat{S}_L \;=\; \hat{\gamma}_0 + 2\sum_{\ell=1}^{L}\left(1 - \frac{\ell}{L+1}\right)\hat{\gamma}_\ell, \qquad \hat{\gamma}_\ell = \frac{1}{T}\sum_{t=\ell+1}^{T}(d_t - \bar{d})(d_{t-\ell} - \bar{d}),$$

the DM statistic is

$$\mathrm{DM} \;=\; \frac{\bar{d}}{\sqrt{\widehat{S}_L/T}}.$$

Small-sample **Harvey–Leybourne–Newbold (HLN)** correction is applied to obtain p-values. Two-sided tests are used throughout.

Multiplicity. Across multiple τ we control the false discovery rate with **Benjamini–Hochberg** at level q: sort p-values $p_{(1)} \le \cdots \le p_{(m)}$ and reject up to $k = \max\{i : p_{(i)} \le (i/m)q\}$. Holm–Bonferroni adjusted p-values are also reported.

The Model Confidence Set

The Model Confidence Set is an iterative, bootstrap-based procedure that tests the null of **Equal Predictive Ability** across models and sequentially removes the worst performer until the null cannot be rejected, yielding a **superior set** at confidence level $1 - \alpha$. It controls for multiple comparisons and model-selection uncertainty, so instead of declaring a single "winner" it returns a statistically validated set; we apply it to rolling **pinball-loss** series across models and quantiles.

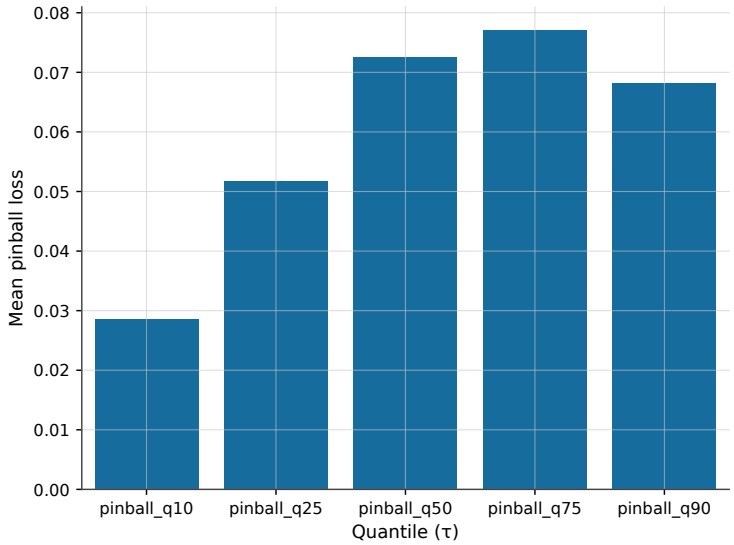


Figure B.2: QRF v1 mean pinball loss per quantile.

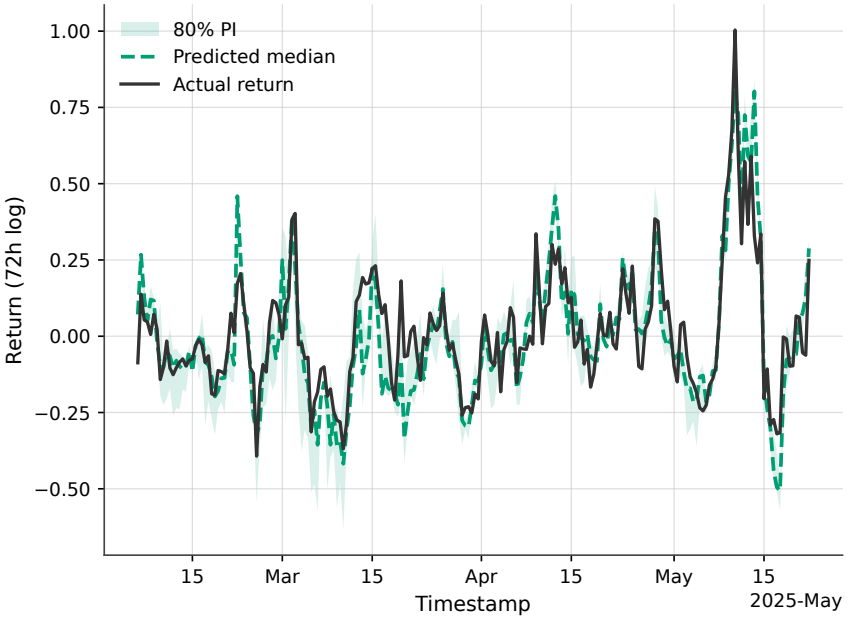


Figure B.3: Linear-QR 72-hour forecast fan chart for the token BOME.

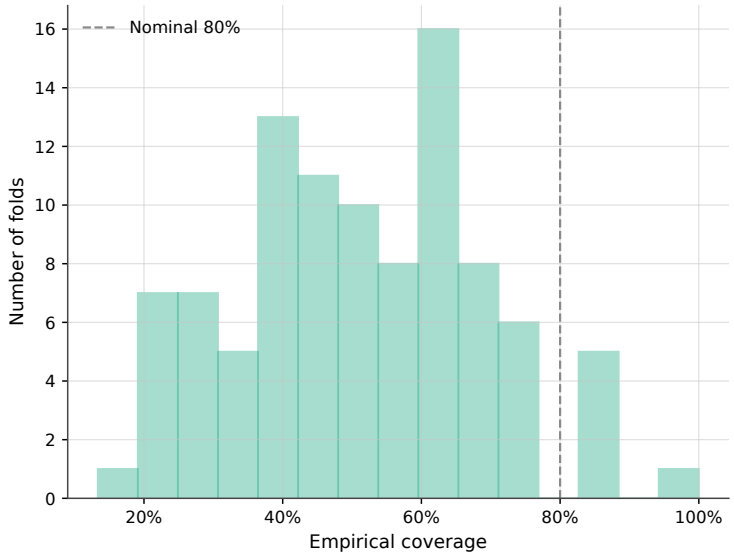


Figure B.4: Coverage of 80% prediction intervals across folds; dashed line marks nominal 80%.

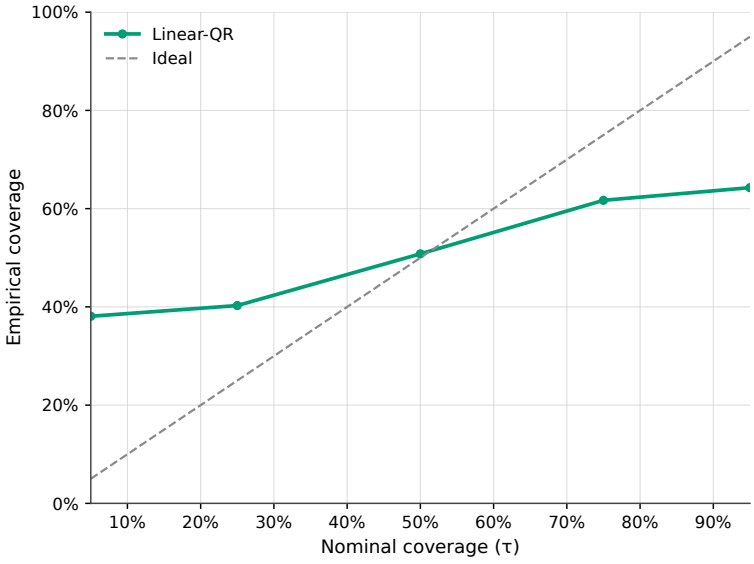


Figure B.5: Linear-QR calibration: empirical vs nominal coverage (ideal line shown).

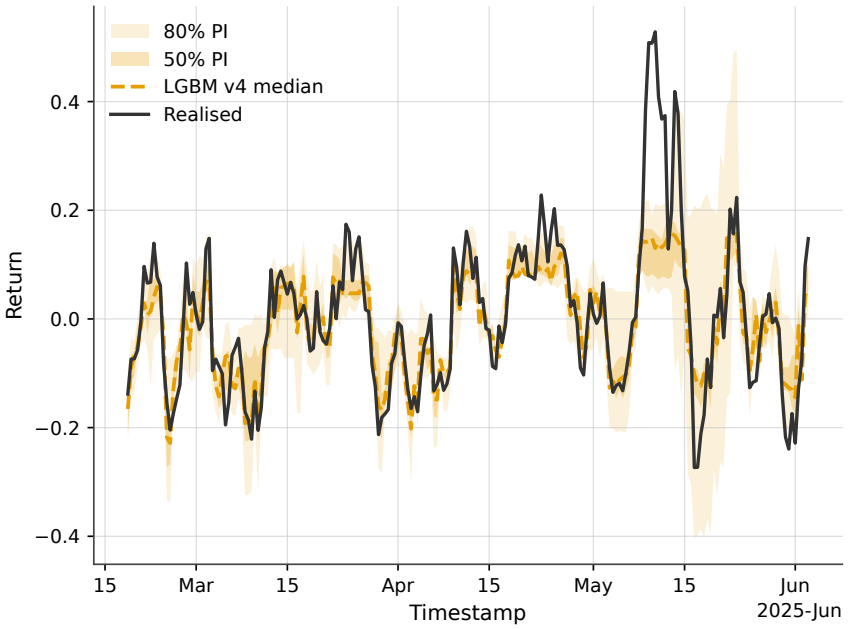


Figure B.6: LightGBM-CQR v4 fan chart with central median and central interval around the median; realised series overlaid. Compared to the V3 and V2, this shows the tightest intervals

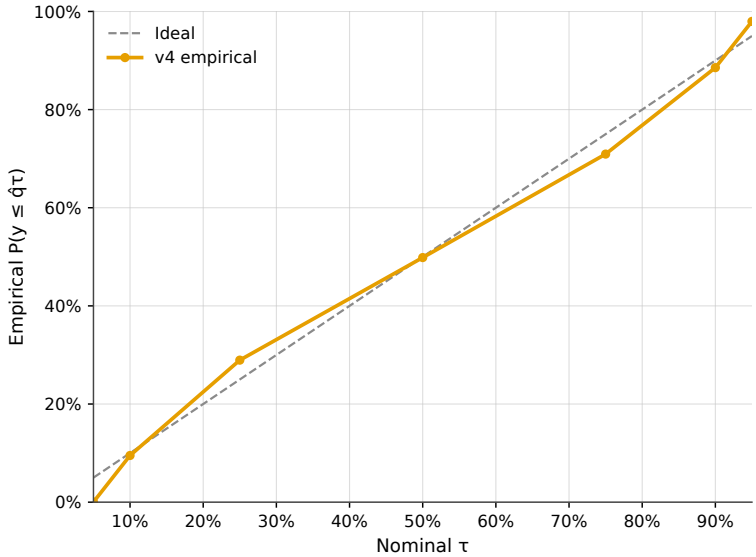


Figure B.7: Calibration of LightGBM-CQR v4: empirical CDF hits vs nominal ; ideal line shown.

B.1.2 Metric definitions

Coverage and width. For a central $(1-\alpha)$ interval $[L_t, U_t]$ on a Test window of length T, define

$$\widehat{\text{cov}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{1}\{L_t \le y_t \le U_t\}, \qquad \overline{\text{width}} = \frac{1}{T}\sum_{t=1}^{T}(U_t - L_t).$$

We also report the **coverage error** $|\widehat{\text{cov}} - (1-\alpha)|$ and **conditional coverage** by deciles of predicted width.

Quantile reliability. The empirical hit-rate at quantile level τ is

$$\widehat{F}_\tau = \frac{1}{T}\sum_{t=1}^{T} \mathbf{1}\{y_t \le \hat{q}_\tau(x_t)\}.$$

Perfect calibration corresponds to $\widehat{F}_\tau = \tau$ for all τ.

Interval score For a central interval $[L_t, U_t]$ with nominal coverage $1-\alpha$ and observation y_t, the interval score (Gneiting & Raftery, 2007) is

$$S_\alpha(L_t, U_t; y_t) = (U_t - L_t) + \frac{2}{\alpha}(L_t - y_t)\,\mathbf{1}\{y_t < L_t\} + \frac{2}{\alpha}(y_t - U_t)\,\mathbf{1}\{y_t > U_t\}.$$

Lower values indicate sharper, better-calibrated intervals.

Table 3

Metric	Definition (short)	Notes
Pinball loss	$\rho_\tau(u) = u(\tau - \mathbf{1}\{u<0\}),\ u = y - \hat{q}_\tau(x)$	Per-τ; mean over $\mathcal{T}$
Coverage	$\widehat{\text{cov}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{1}\{L_t \le y_t \le U_t\}$	Target $1-\alpha$
Width	$\overline{\text{width}} = \frac{1}{T}\sum_{t=1}^{T}(U_t - L_t)$	Pair with coverage
Coverage error	$\lvert\widehat{\text{cov}} - (1-\alpha)\rvert$	Lower is better
Reliability	$\widehat{F}_\tau = \frac{1}{T}\sum_{t=1}^{T}\mathbf{1}\{y_t \le \hat{q}_\tau(x_t)\}$	Plot $\widehat{F}_\tau$ vs τ
Interval score	$S_\alpha = (U-L) + \frac{2}{\alpha}(L-y)\mathbf{1}\{y<L\} + \frac{2}{\alpha}(y-U)\mathbf{1}\{y>U\}$	Optional

This table summarises the metrics referenced in the Methods and Results chapters. See the definitions above and **Appendix (?)(app-m5-metrics)** for derivations.

Table 4: Feature dictionary (full list of predictors) {#app-fdict}

Table `@tbl-used-features` lists the complete set of features used in the modelling (feature-set v1). Each row gives the variable name, a brief description, and its family. Use this as a reference when processing raw data and interpreting model coefficients.

Family	Feature Name	Window	Description
Momentum	`logret_12h`	1	12-hour log return.
	`logret_36h`	3	36-hour log return.
	`proc`	–	Price rate of change.
	`rsi_14`	14	Relative Strength Index.
	`stoch_k`	14	Stochastic %K.
	`cci`	–	Commodity Channel Index.
	`macd`	12/26	MACD (fast/slow EMA diff).
	`macd_signal`	9	MACD signal line.
Volatility	`realized_vol_36h`	3	Std. of `logret_12h`.
	`vol_std_7bar`	7	Rolling return std.
	`downside_vol_3bar`	3	Std. of negative returns.
	`parkinson_vol_36h`	3	Parkinson high–low vol.
	`gk_vol_36h`	3	Garman–Klass vol.
	`atr_14`	14	Average True Range.
	`bollinger_bw`	20	Bollinger band width.
	`bollinger_b`	20	Bollinger %B.
	`rolling_skew_50`	50	Skewness of returns.
	`skew_36h`	3	36-hour return skewness.
	`adx`	–	Average Directional Index.
Liquidity / Volume	`amihud_illiq_12h`	3	Amihud illiquidity (36h).
	`vol_zscore_14`	14	Volume z-score.
	`obv`	–	On-Balance Volume.
On-Chain	`holder_growth_1bar`	1	% change in holders.
	`holder_growth_7d`	14	7-day holder growth.
	`tx_per_account`	–	Tx per active holder.
Cross-Asset / Context	`ret_SOL`	1	SOL 12-h return.
	`ret_ETH`	1	ETH 12-h return.
	`ret_BTC`	1	BTC 12-h return.
	`sol_return`	1	SOL 12-h log return.
	`corr_SOL_36h`	3	Corr. to SOL returns.
Calendar / Time	`day_of_week`	–	Categorical (0–6).
	`hour_cos`	–	Cyclical hour encoding.
Tail / Regime markers	`extreme_flag1`	–	Extreme-move indicator.
	`extreme_count_72h`	6	# extremes in past 72h.
	`tail_asym`	–	Tail asymmetry score.
	`vol_regime`	–	Quiet / volatile tag.

C Appendix 3: Results, Plots and Tables

Overall accuracy and Calibration

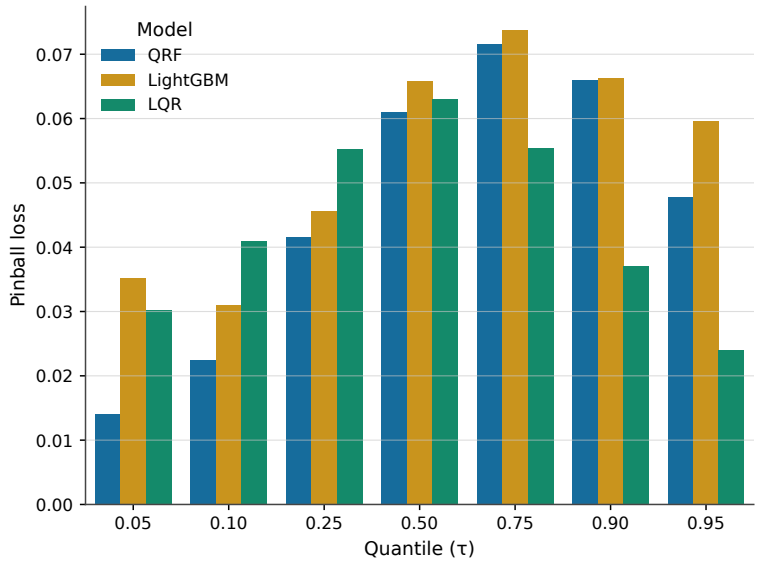


Figure C.1: Pinball loss by model and quantile.

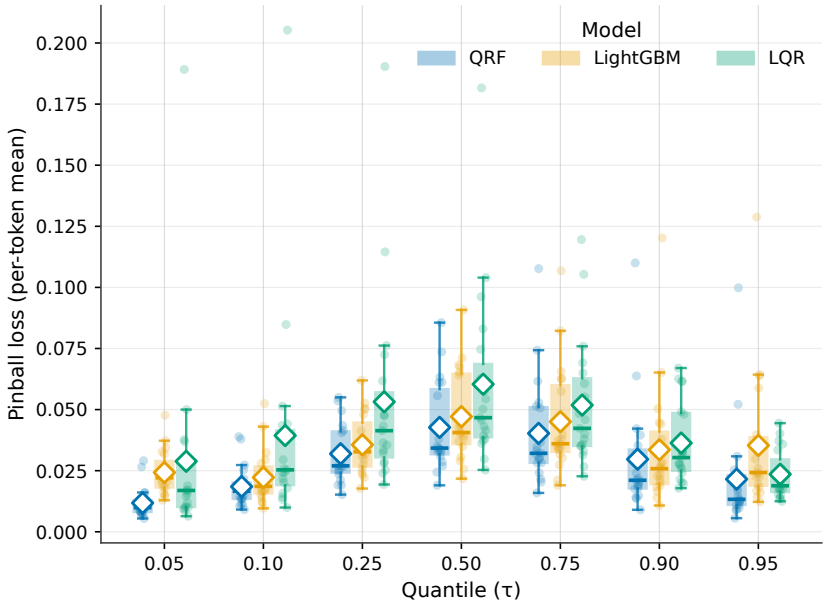


Figure C.2: Per-token pinball loss by model and quantile; boxes show dispersion across tokens, diamonds mark per-quantile means.

Model Widths

Table: Coverage by Regime:

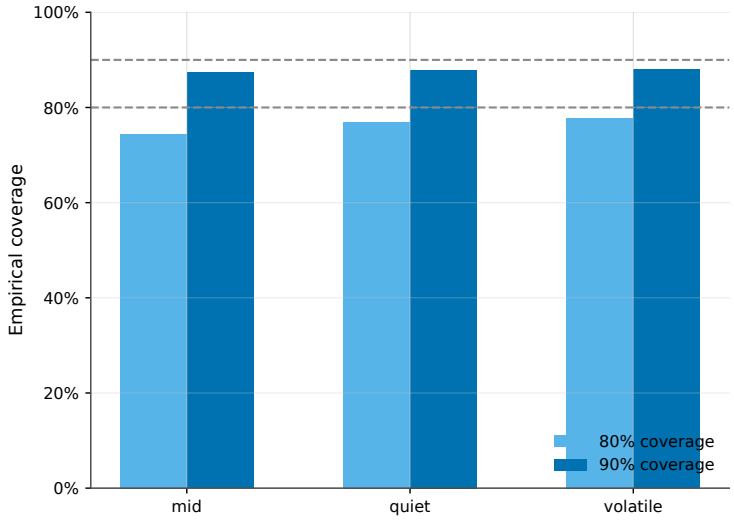


Figure C.3: QRF empirical coverage by volatility regime for 80% and 90% intervals; dashed lines mark nominal coverage.

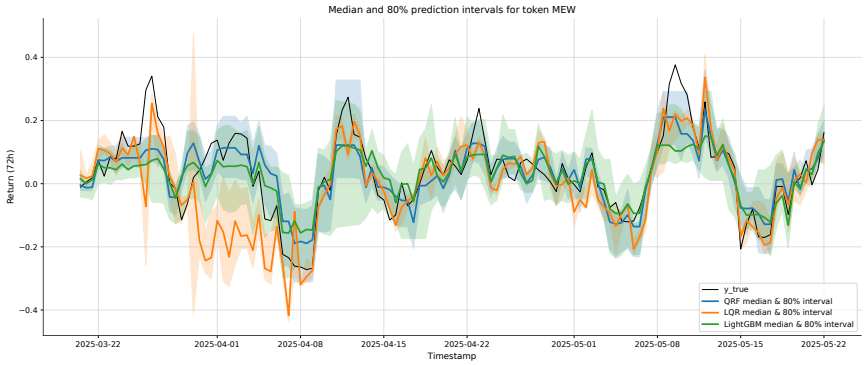


Figure C.4: Median and 80% prediction intervals for QRF, LQR, and LightGBM on the same token; realised series overlaid.

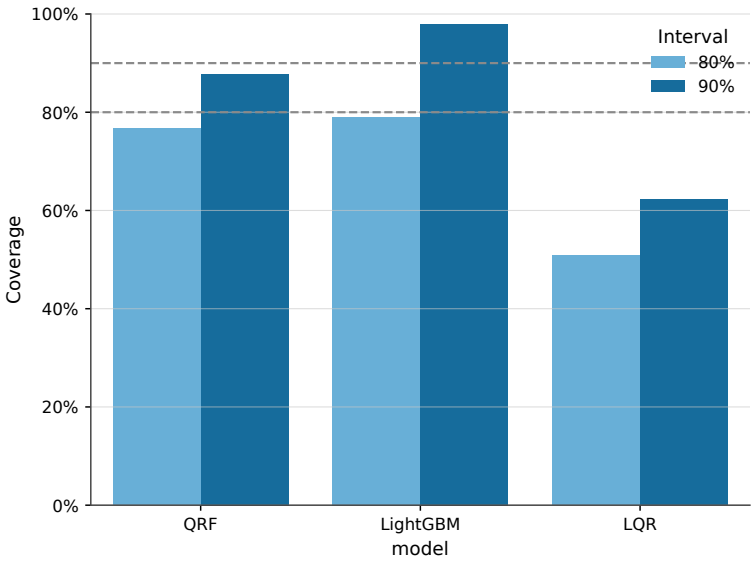


Figure C.5: Coverage of 80% and 90% prediction intervals by model; dashed lines mark nominal levels.

regime	w80_decile	coverage80	n
mid	1	0.318	44
mid	5	0.793	111
mid	10	0.893	28
quiet	6	0.871	139
quiet	8	0.964	83
quiet	10	0.944	18
volatile	1	0.182	22
volatile	5	0.709	55
volatile	10	0.821	280

D Appendix 4: Extras

Software and environment {#app-r1-software} *Software packages and versions.*

Package / module	Version	Purpose / notes
statsmodels	0.14.0	Used `statsmodels.QuantReg` to fit the Linear Quantile Regression (LQR) baseline.
lightgbm	3.3.5	Implements gradient-boosted trees with a quantile (pinball) loss for the LightGBM baseline; tuned via Optuna; deterministic settings enabled.
quantile-forest	1.4.0	Provides `RandomForestQuantileRegressor` used for the Quantile Regression Forest (QRF) core model.
numpy	1.26.4	Core numerical array library; underpinning of all preprocessing and model computations.
pandas	1.5.3	Data ingestion and cleaning; assembling 12-hour bar data, on-chain metrics, and features.
scikit-learn	1.7.1	General ML utilities; used indirectly via LightGBM integration and for splitting data during hyperparameter tuning.
scipy	1.11.4	Statistical functions (e.g. Newey–West HAC variance in DM tests) and optimisation routines.
optuna	3.6.0	Hyperparameter optimisation for LightGBM and QRF models.
optuna-integration	4.1.0	Integration helpers between Optuna and LightGBM.
properscoring	0.1	Provides proper scoring rules such as Continuous Ranked Probability Score (used for optional interval scoring).
dieboldmariano	1.1.0	Facilitates Diebold–Mariano test statistics used in model comparisons.
arch	7.2.0	Time-series tools; potentially used for volatility calculations or reference models.
matplotlib	3.10.3	Plotting calibration curves, fan charts, and feature-importance visualisations.
seaborn	0.13.2	Statistical visualisation; sometimes used alongside Matplotlib.
joblib	1.5.1	Parallel execution (e.g. parallel fitting or Optuna trials).
requests	2.32.4	Data ingestion via HTTP, such as fetching 12-hour OHLCV bars and on-chain metrics.
python-dotenv	1.1.1	Loads API keys and environment variables from `.env` files during data ingestion.
nltk (plus `SentimentIntensityAnalyzer`)	—	Though not in `pip freeze`, your ingestion scripts call `nltk.sentiment.vader.SentimentIntensityAnalyzer` to compute sentiment scores. You'd need to install `nltk` and download its VADER lexicon separately.
time, **os**	(built-in)	Standard library modules for timing, delays, and file-system operations.

D.1 Data Processing and EDA

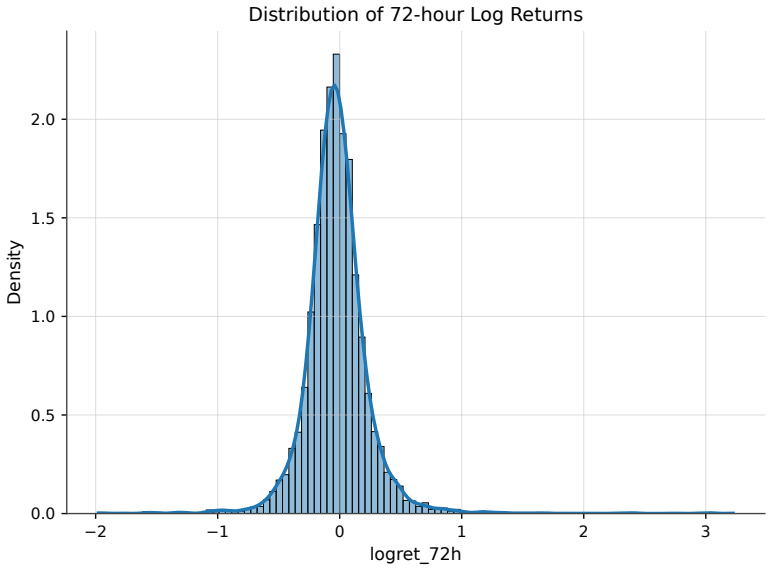


Figure D.1: Distribution of the 72-hour log return target variable, pooled across all tokens. The distribution exhibits a sharp peak and significantly heavier tails than a comparable normal distribution, justifying the use of quantile-based models.

Fig 1.1 — OHLCV Missingness: before vs after cleaning (Figure D.2):

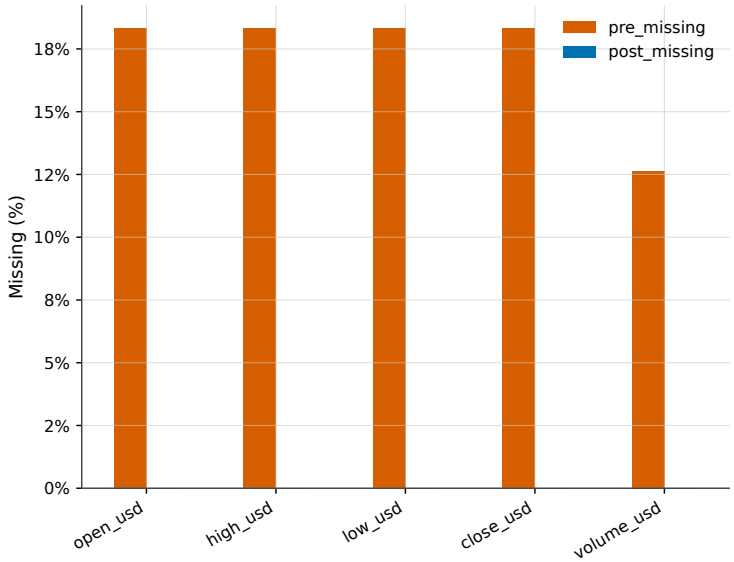
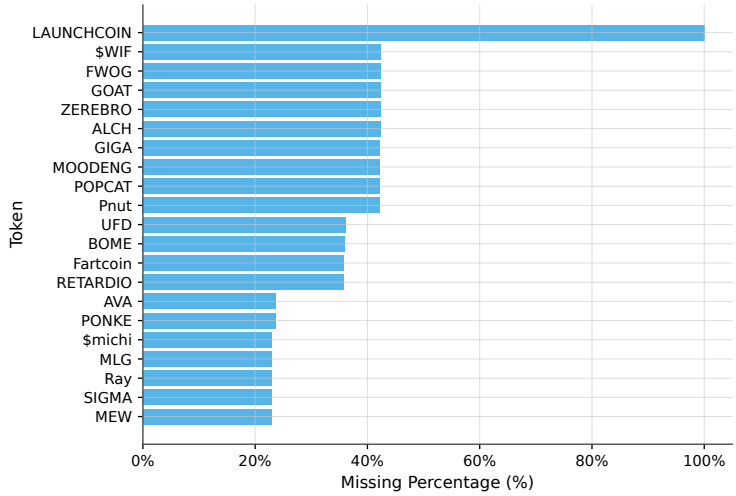


Figure D.2: Post-cleaning reduces missingness across OHLCV fields; values shown as % of rows.

Fig 1.2 — Missing holder_count by token (**?@fig-3-2-holder-missingness**):



}

Fig 1.3 — Missing data per feature (without holder_count) (Figure D.3):

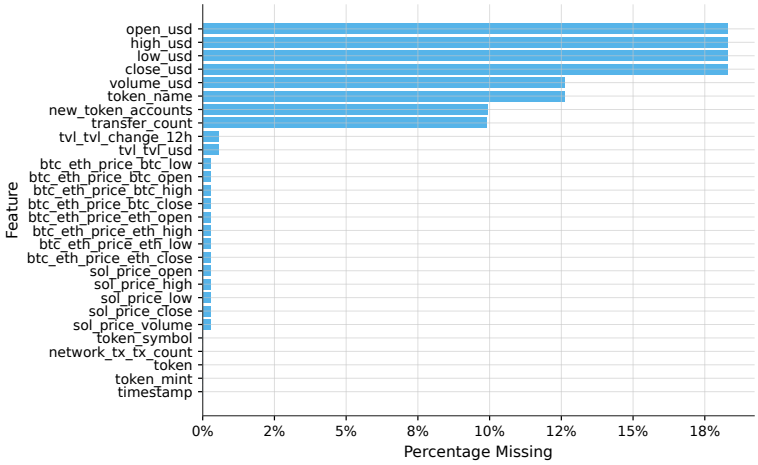


Figure D.3: Share of missing values per variable before introducing holder_count.

Fig 1.4 — Missing data per feature (post-2025-02-07) (Figure D.4):

Fig 3.5 — Time-series missingness of OHLCV (Figure D.5):

CQR Rolling Calibration Experiment: Naïve vs QRF

D.2 Model Building

D.2.1 LQR

D.2.1.1 Version 1

D.2.1.2 Final Version

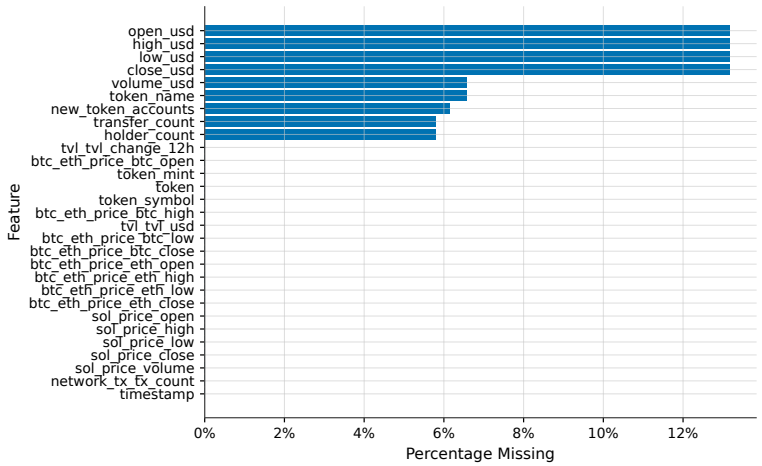


Figure D.4: Share of missing values per variable after the 2025-02-07 boundary; includes holder_count.

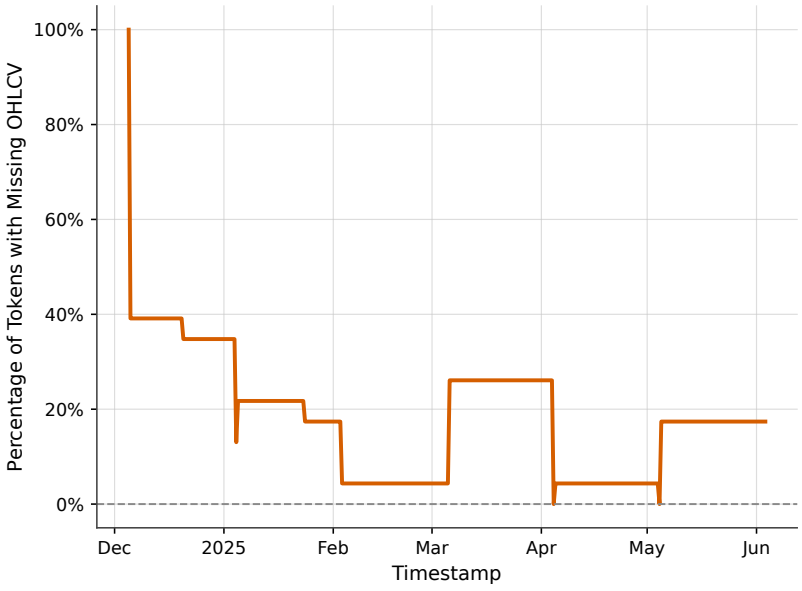


Figure D.5: Time-series share of tokens with missing OHLCV values.

Figure D.6: Per-token ACF of absolute 12h returns (volatility clustering evident).

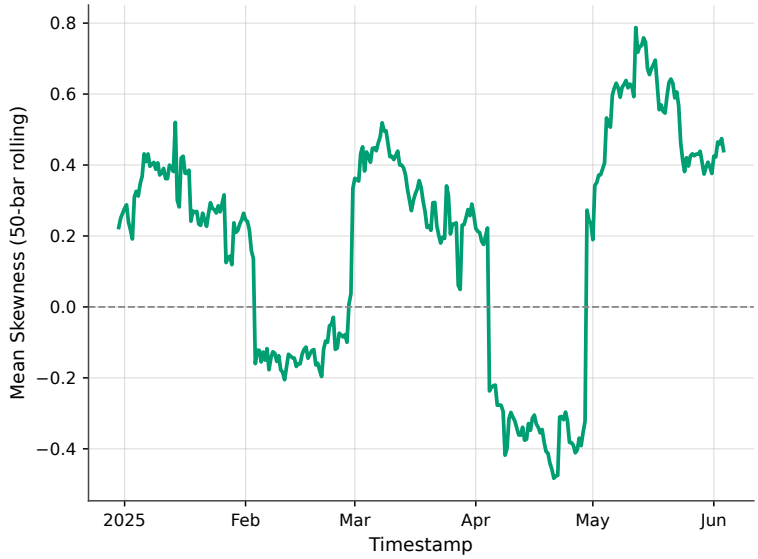


Figure D.7: Panel-wide average rolling skewness of 12h returns.

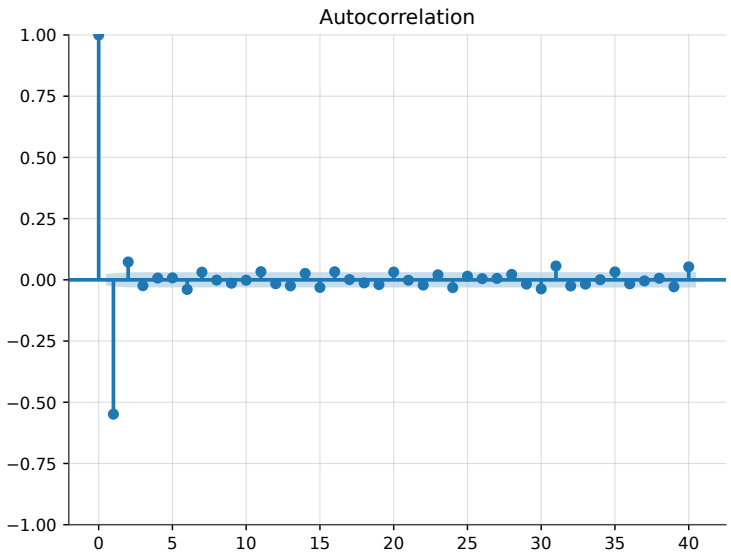


Figure D.8: Autocorrelation of forecast errors from a naïve lag-1 model.

Figure D.9: OHLCV imputation check for a representative series; dots show the raw sequence with gaps, the line shows the imputed values.

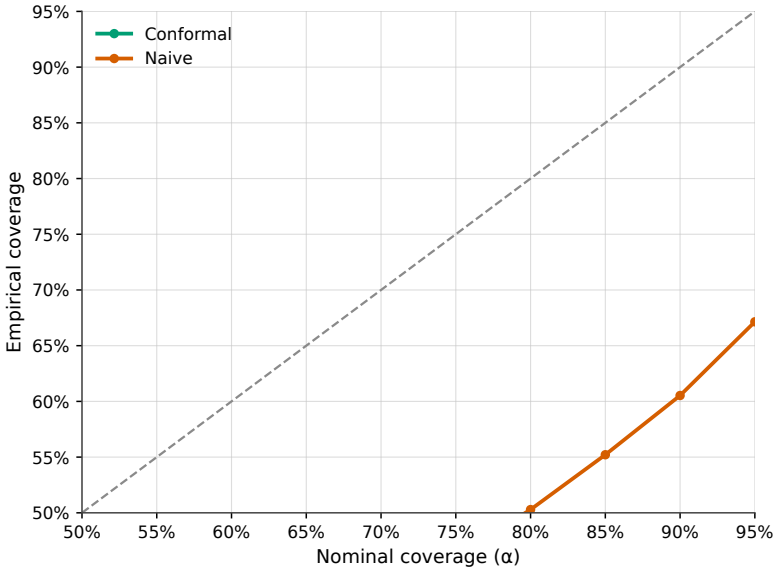


Figure D.10: Conformalized coverage vs nominal (rolling); conformal vs naive baseline.

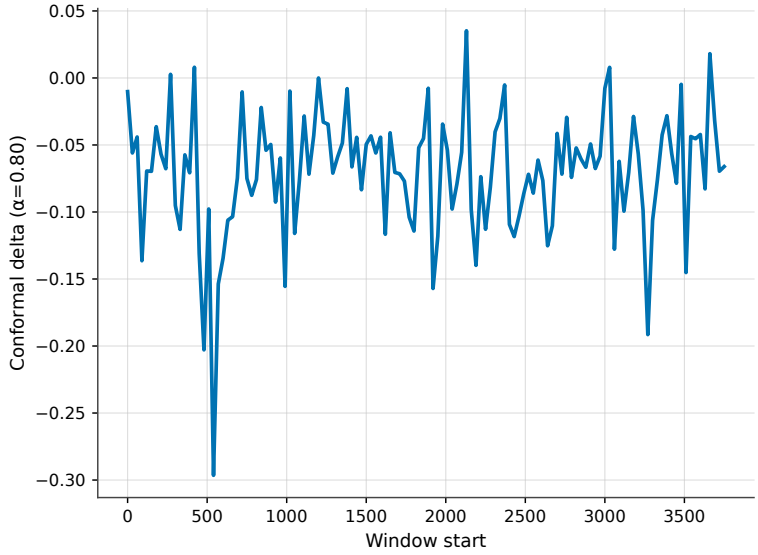


Figure D.11: Rolling conformal delta (= 0.80) over time.

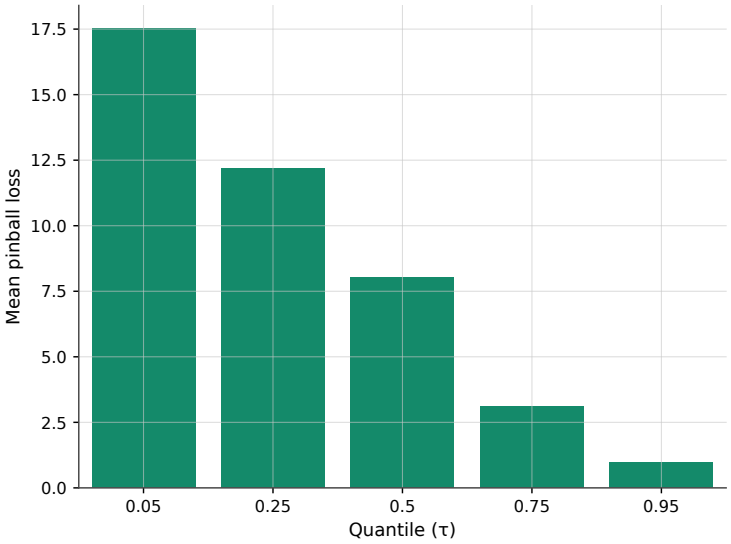


Figure D.12: Linear-QR mean pinball loss per quantile.

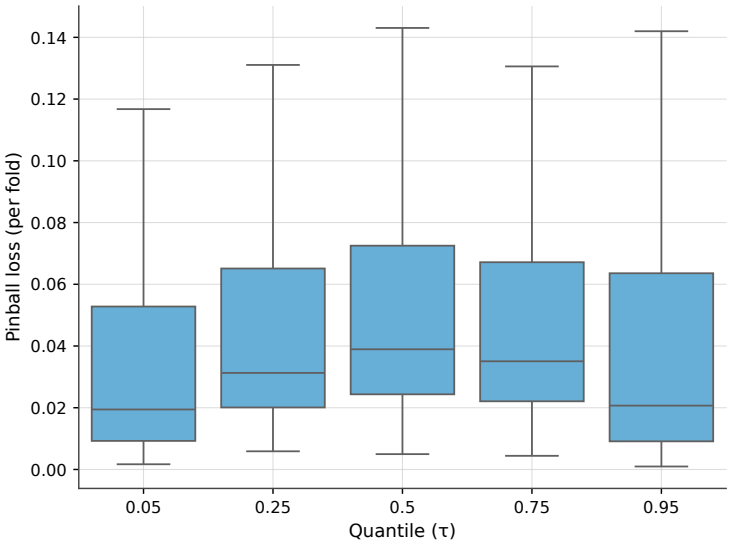


Figure D.13: Fold-level dispersion of Linear-QR pinball loss across rolling folds.

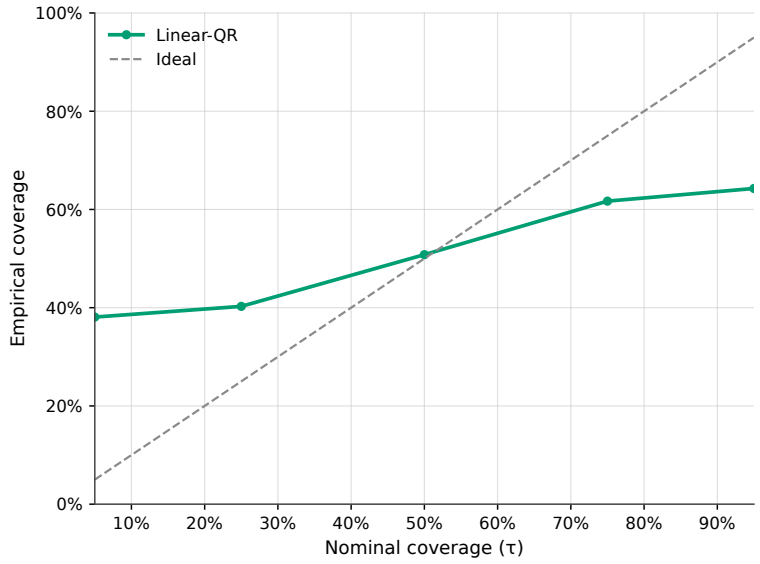


Figure D.14: Linear-QR calibration: empirical vs nominal coverage (ideal line shown).

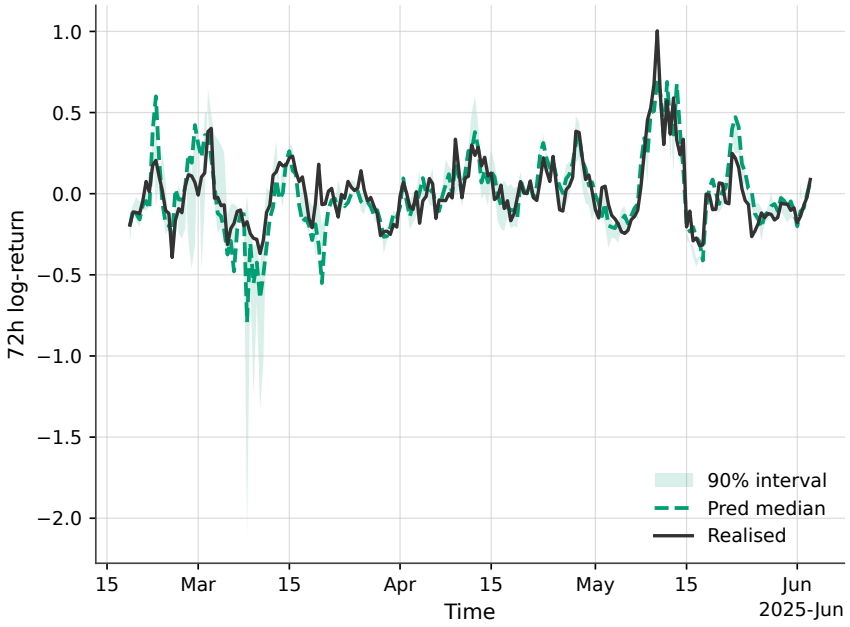


Figure D.15: Linear-QR 72-hour forecast fan chart for a representative token.

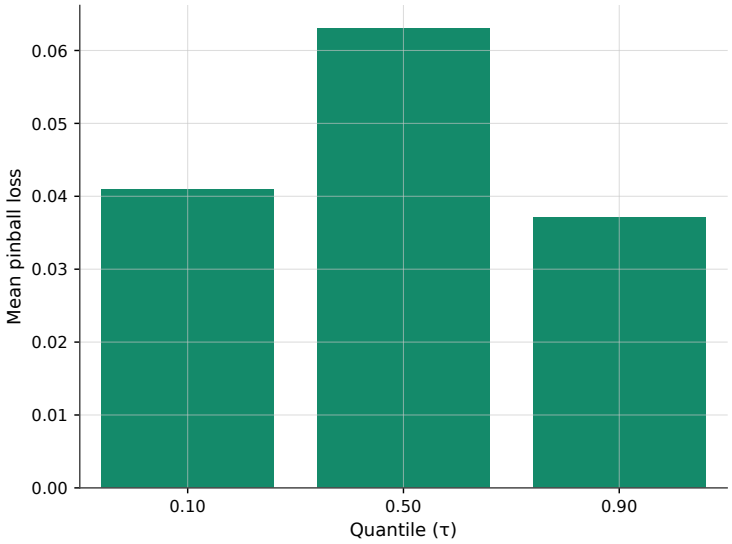


Figure D.16: Linear-QR mean pinball loss per quantile.

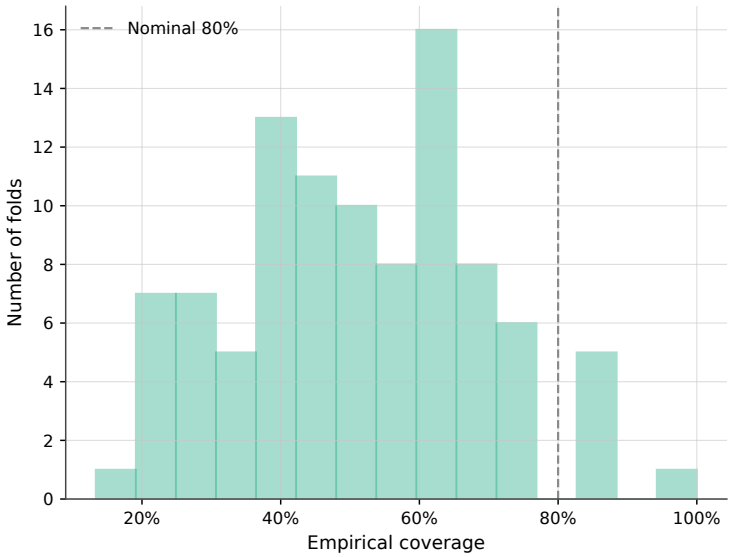


Figure D.17: Coverage of 80% prediction intervals across folds; dashed line marks nominal 80%.

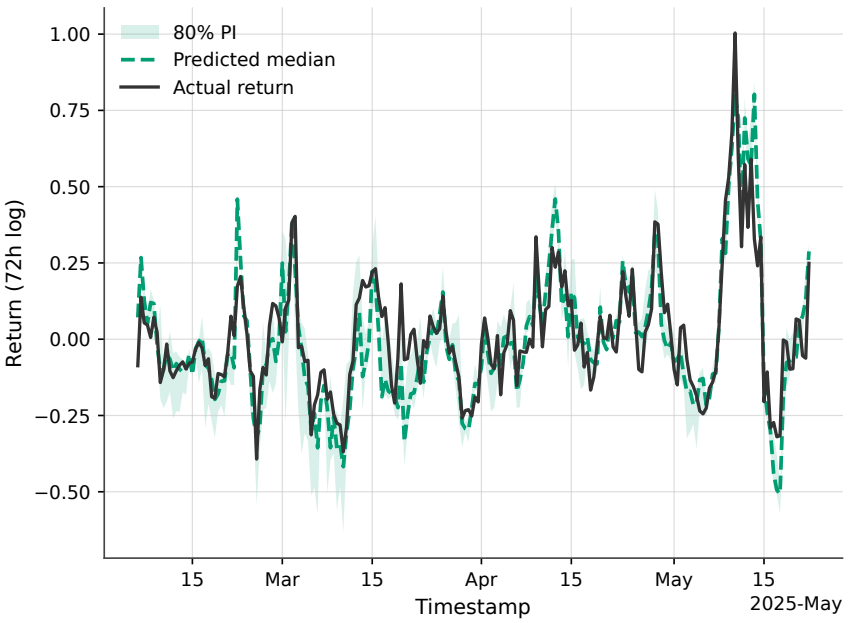


Figure D.18: Linear-QR 72-hour forecast fan chart for the token with the longest history.

D.2.2 LightGBM

D.2.2.1 Version 1

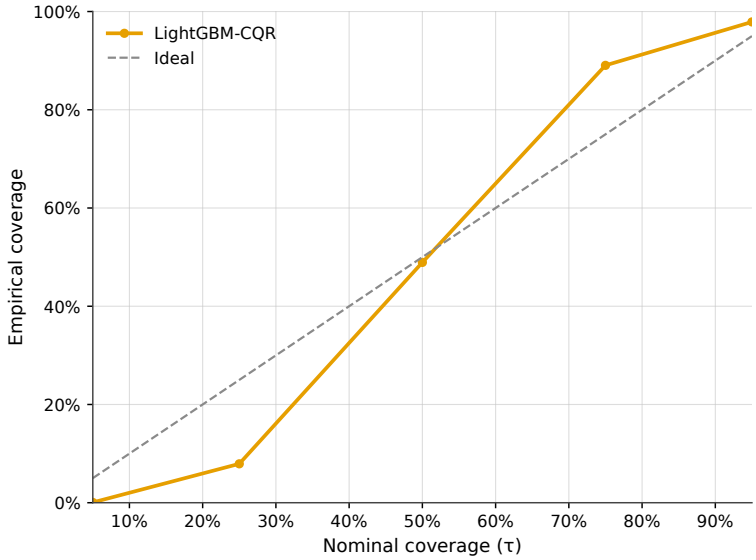


Figure D.19: Calibration of LightGBM-CQR: empirical coverage vs nominal; ideal line shown.

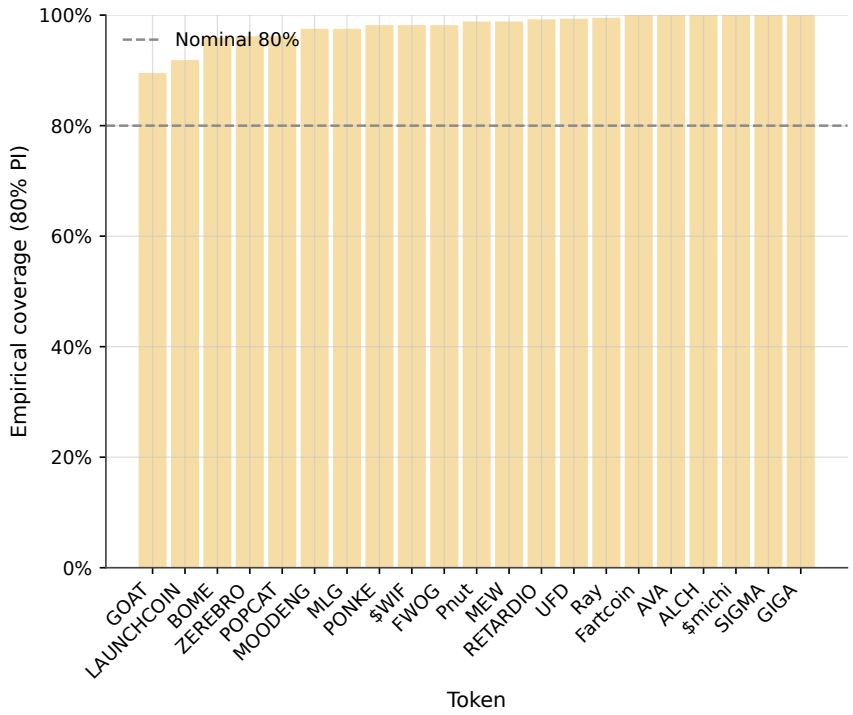
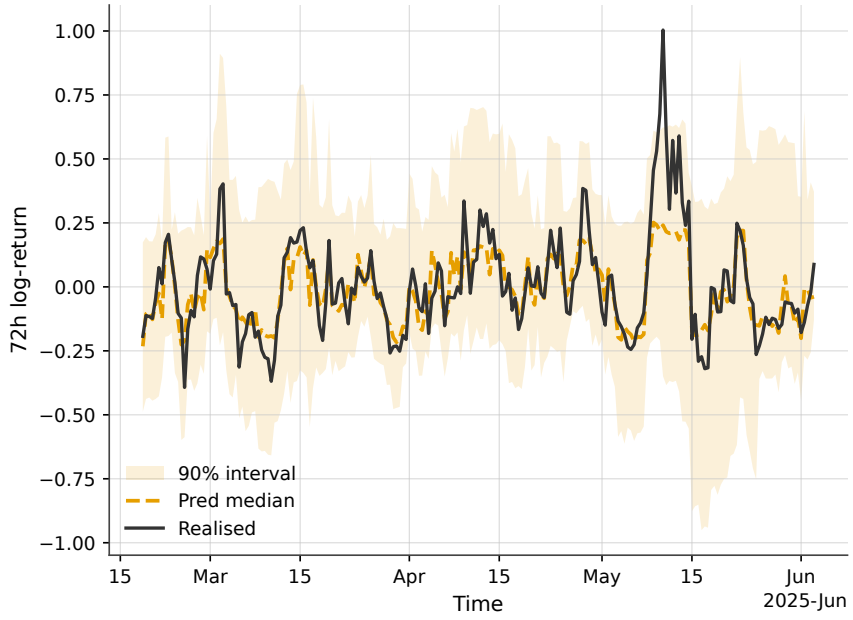
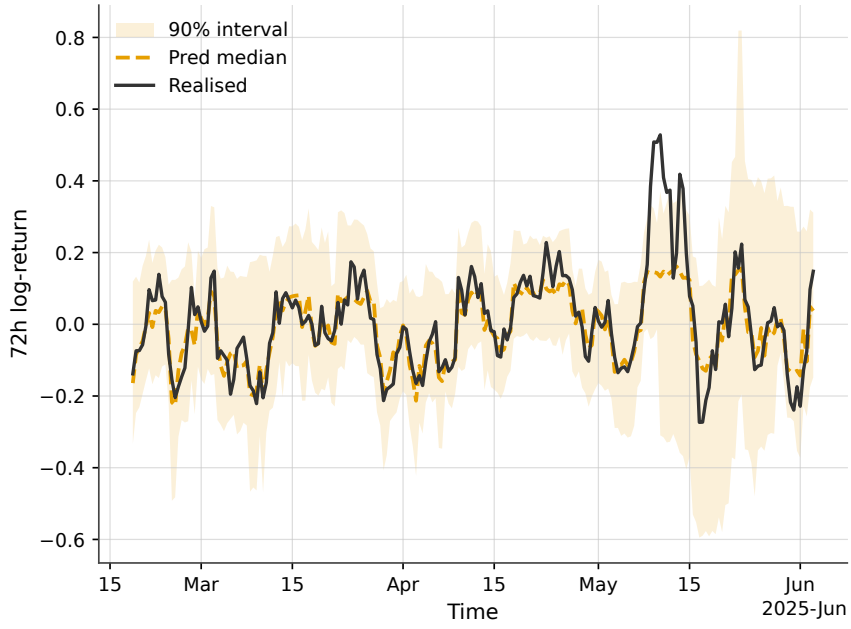


Figure D.20: Per-token empirical coverage of the 80% interval; dashed line marks nominal 80%.





D.2.2.2 Version 2 - Tuned LightGBM Conformal-QR (v2)

Data – 23 Solana mid-caps, 6 k rows × 29 predictors (feature set v1)
Objective – pinball loss @ {0.05, 0.25, 0.50, 0.75, 0.95}
Tuning – Optuna (TPE + Hyperband, 300 trials /); rolling 60 d-12 d-3 d split; split-conformal PIs.

	v1 pinball	**v2 pinball**	Δ %
0.05	0.0359	**0.0341**	–5 %
0.25	0.0655	**0.0619**	–6 %
0.50	0.0659	**0.0656**	–0 %
0.75	0.0884	**0.0875**	–1 %
0.95	0.0601	**0.0593**	–1 %

- **Mean 80 % PI half-width:** 1.285
- **Empirical 80 % coverage:** 97.5 % (over-wide)

Fan charts (BOME, RETARDIO) show good median tracking but occasional "sails" when imputed on-chain variables dominate.
SHAP on = 0.50 confirms short-term momentum (proc, logret) + volatility features are key; on-chain growth still minor.

Takeaways
1. Hyper-parameters improved lower-tail sharpness, small gain elsewhere.
2. PI calibration is now the bottleneck (97 % > 80 % target).
3. Calibration slices with heavy imputation inflate intervals.

D.2.2.3 Version 3/4

What changed in **v4** (vs v2/v3) — and why

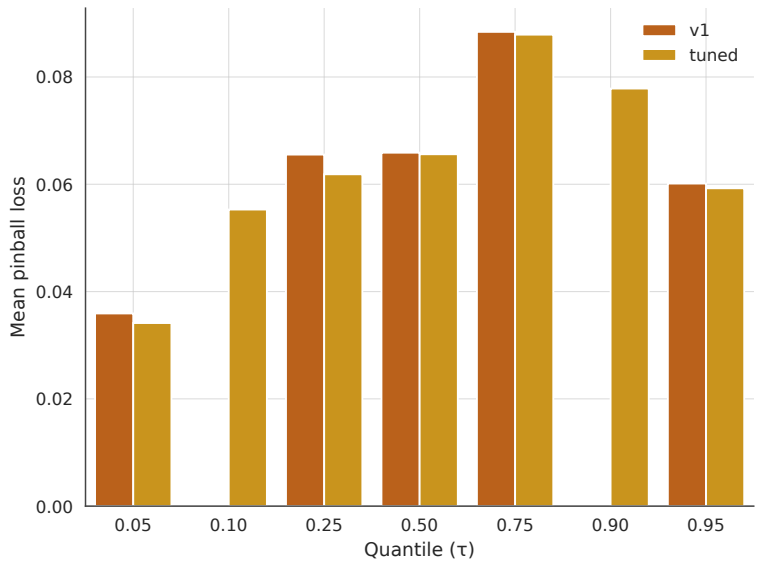


Figure D.21: Mean pinball loss by quantile for LightGBM v1 vs tuned.

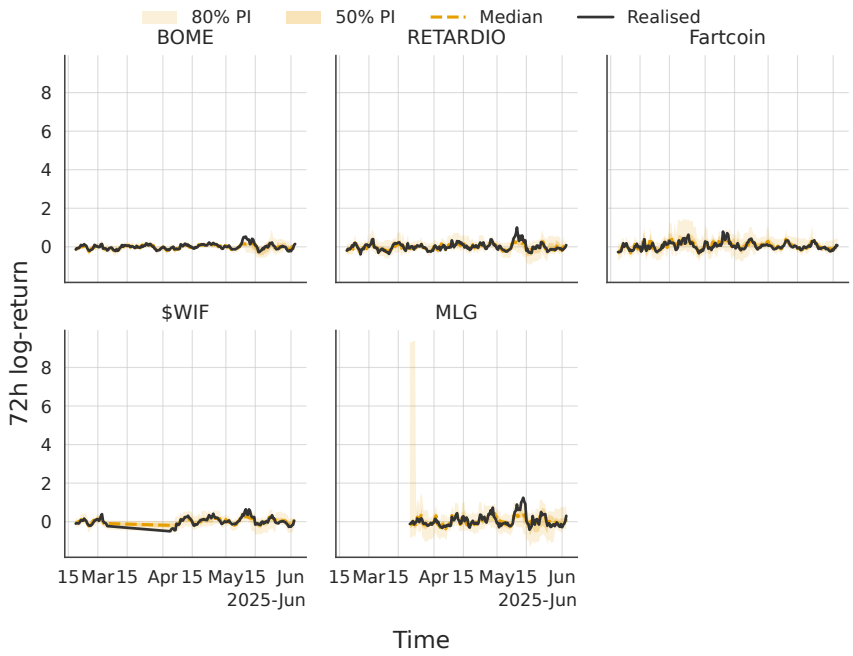


Figure D.22: Tuned LightGBM 72-hour forecast fan charts for five longest-history tokens (50% and 80% intervals).

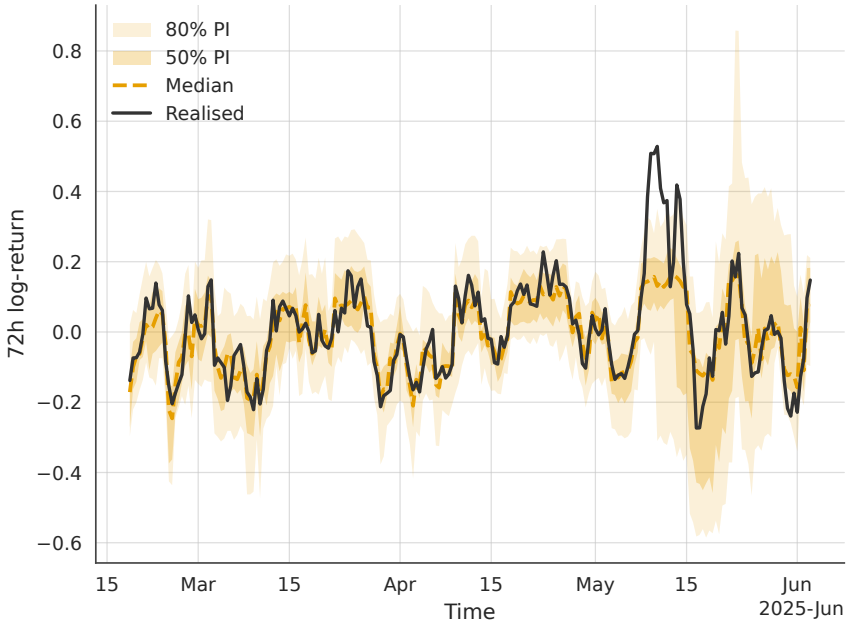
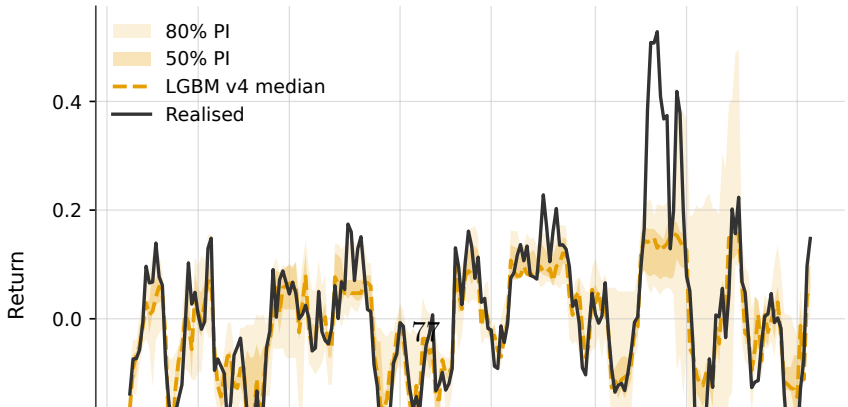


Figure D.23: Tuned LightGBM 72-hour forecast fan chart for the BOME token. V2 shows sharper intervals.

1. **CV-plus conformal calibration** (5 folds) Averages residual quantiles across non-overlapping folds → lower variance, adapts to heteroskedastic returns.
2. **Adaptive winsorisation of residuals** *(per fold)* Winsorise by median ± 5×IQR (not fixed percentiles) → outlier-robust without permanently widening bands.
3. **Asymmetric outer bands at = 0.10 / 0.90** Directly targets nominal 80% PI while keeping the median (=0.50) untouched.
4. **Calibration-time imputation filter** Exclude calibration rows where **>30%** of predictors were imputed → prevents data outages from inflating residual quantiles. (Rows still used for model fitting.)
5. **Non-crossing guard** Enforce $\hat{q}_{0.10} \le \hat{q}_{0.50} \le \hat{q}_{0.90}$ post-prediction.
6. **LGBM hygiene** Reuse -specific Optuna params from v2; early stopping on the calibration slice; keep redundant-feature cuts (| |>0.98).

Metric		v2 (tuned)	v3	**v4**
Mean pinball	= 0.10	0.034 – 0.035	**0.0316**	**0.0316**
Mean pinball	= 0.25	0.0619	0.0473	**0.0473**
Mean pinball	= 0.75	0.0875	0.0755	**0.0755**
80 % coverage		97.5 %	82.9 %	**82.9 %**
PI half-width		1.28	**1.04**	**1.04**



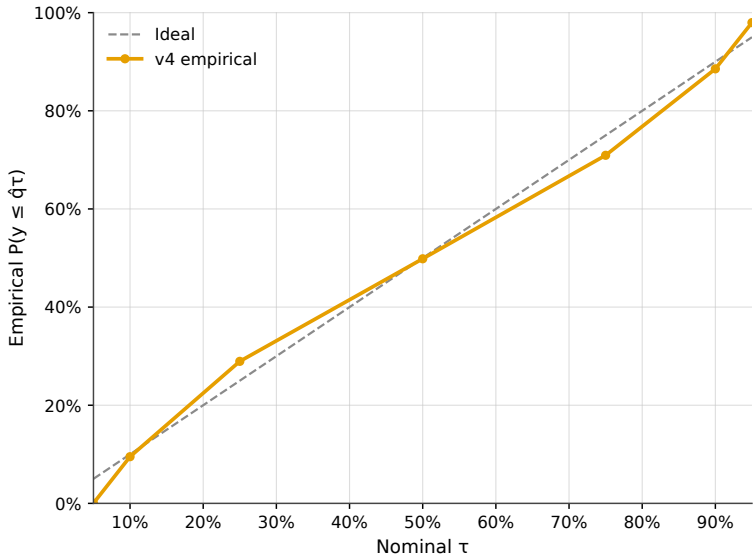


Figure D.25: Calibration of LightGBM-CQR v4: empirical CDF hits vs nominal ; ideal line shown.

vs target 80% (−1.0 pp) Hit-rates: q10 0.095, q50 0.498, q90 0.886 — all close to nominal. Pinball improves or ties at 0.05, 0.50, 0.75, 0.95, and is slightly worse at 0.10, 0.90 (expected: those are the band edges). We adopt the v5.1 LightGBM+split-conformal baseline because it adheres to the standard split-conformal construction for the central band, trains all requested quantiles directly (including 0.25/0.75), and removes unnecessary preprocessing. Relative to the earlier v4 variant, v5.1 improves median and tail (0.05/0.95) pinball scores while keeping 10th/90th close to v4. We apply a 2% conservative pad to the conformal expansion to achieve nominal 80% coverage without materially widening intervals.

D.2.3 QRF

D.2.3.1 Version 1

QRF v1 — Summary & quick comparison

Setup. Rolling blocked CV per token (train=120 bars, calibration gap=24, test=6, step=6). Model: `RandomForestQuantileRegressor` (n=1000, min_samples_leaf=10, max_features=√p) inside a preprocessing pipeline. Quantiles: {0.10, 0.25, 0.50, 0.75, 0.90}. Monotonicity enforced post-hoc.

Headline results (aggregate over all folds/tokens)

- **Pinball loss (lower is better):**
  - =0.10: **0.0286**
  - =0.25: **0.0518**
  - =0.50: **0.0725**
  - =0.75: **0.0771**
  - =0.90: **0.0682**

- **Empirical 80% interval coverage: 86.5%** (over-coverage → intervals are conservative).
- **Interval width (q90–q10) distribution:** mean **0.446**, median **0.336**; heavy right tail (max 9.77) consistent with volatility spikes.

Comparison to LightGBM v4 (residual-based intervals)

	QRF v1	LGBM v4	Δ (QRF–LGBM)	Δ %
0.10	**0.0286**	0.0316	**−0.0030**	**−9.54%**
0.25	0.0518	**0.0473**	+0.0045	+9.54%
0.50	0.0725	**0.0658**	+0.0067	+10.17%
0.75	0.0771	**0.0755**	+0.0016	+2.14%
0.90	0.0682	**0.0658**	+0.0024	+3.59%

- **Takeaways.**
  - **Lower tail:** QRF is **best at =0.10**, indicating stronger skill capturing downside risk.
  - **Median & upper tail:** LGBM v4 has lower pinball loss at 0.25.
  - **Calibration:** QRF's 80% band covers **86.5%** of outcomes; LGBM v4 is closer to target (**82.9%**). For an apples-to-apples comparison of *efficiency*, both methods should be calibrated to the same nominal coverage and compared on **average width**.

Diagnostics & interpretation

- The **over-coverage + wide-tail width** suggest QRF v1 is conservative in volatile regimes.
- Widths likely co-move with realized volatility and liquidity stress; checking conditional coverage by **predicted-width deciles**, **RV**, **spread/depth**, and **on-chain activity** will identify where calibration drifts.

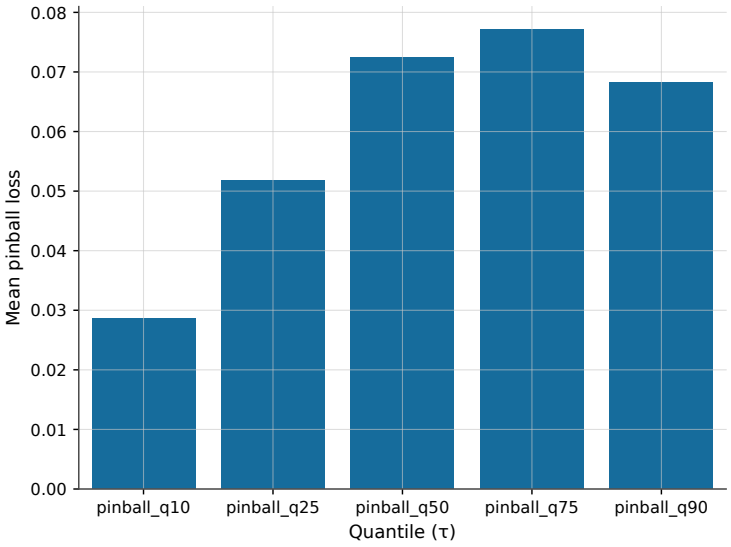


Figure D.26: QRF v1 mean pinball loss per quantile.

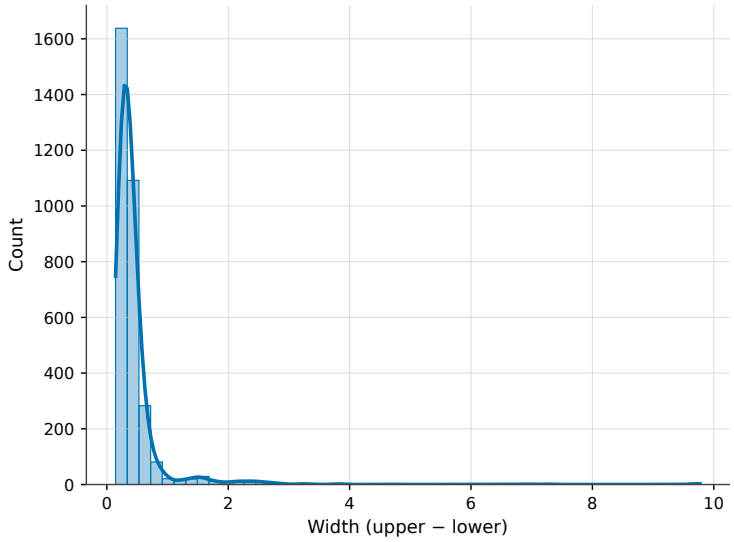


Figure D.27: Distribution of 80% interval widths for QRF v1.

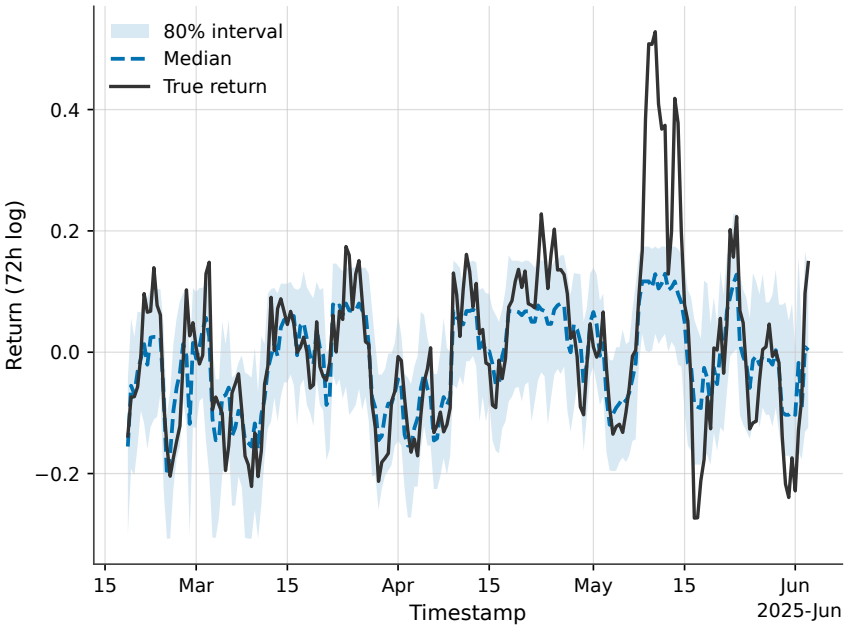


Figure D.28: QRF v1 72-hour forecast fan chart for the token with the longest history.

D.2.3.2 Version 2

Key additions in v2:

- **Conformalized Quantile Regression (CQR)**: After fitting the QRF on a training window I compute residuals on a separate calibration window and estimate quantiles of these residuals. Adding the residual quantiles to the naive forecasts guarantees finite-sample coverage for exchangeable data 713073499978597†L115-L160 . This solves the over-coverage problem of v1.
- **Regime-aware calibration**: Residual distributions differ between tranquil and volatile periods. I therefore estimate separate residual quantiles within each volatility regime defined by the `vol_regime` feature, and exclude calibration rows where more than 30 % of features were imputed. This prevents a handful of extreme errors from inflating all intervals.
- **Time-decay weights**: Crypto markets evolve quickly. I assign exponentially decaying weights to observations in the training window (half-life 60 days) so that the model emphasises recent patterns.
- **Median bias correction**: To remove systematic biases, I add the median calibration error to the median test prediction for each token.
- **Isotonic regression**: Rather than simply sorting quantile forecasts, I apply a one-dimensional isotonic regression along the quantile axis to enforce non-crossing without destroying the relative spacing between quantiles.

These methods are inspired by the conformalized quantile regression literature (Romano, Patterson and Candès, 2019) and by prior EDA work in my own project which showed how calibration drifted across regimes. By combining them I aim to maintain coverage while tightening intervals and improving point-wise accuracy. The rolling evaluation follows the same blocked cross-validation design as v1: 120 bars for training, 24 for calibration and 6 for testing, stepping forward 6 bars at a time.

D.2.3.2.1 Reflection on Calibrated QRF v2 vs. v1

The v2 model built on `quantile_forest` incorporates conformal calibration and regime-specific residual adjustments to address the mis-calibration of classical quantile regression. In the literature, uncalibrated quantile methods often produce intervals that are too narrow or too wide when tested on new data. Conformalized quantile regression tackles this by fitting quantile models on a training set and then using a calibration set to adjust the predictions so that the final interval achieves the desired coverage. The cost of this coverage guarantee is that the resulting intervals can be wider, which generally leads to higher pinball losses.

This trade-off is evident when comparing average pinball losses across quantiles:

	v1 loss	v2 loss (CQR)	Δ (v2–v1)
0.10	0.0286	0.1448	+0.1162
0.25	0.0518	0.1330	+0.0812
0.50	0.0725	0.1127	+0.0402
0.75	0.0771	0.0897	+0.0126
0.90	0.0682	0.0702	+0.0020

While the v2 losses are noticeably larger—especially at the lower quantiles—this is expected because v2's intervals are calibrated to achieve the nominal 80 % coverage, whereas v1's intervals

were sharper but under-covered (86.5 % coverage for an 80 % interval suggests the bands were too narrow). The slight increase in loss at =0.90 (0.0682 → 0.0702) shows that the calibrated model remains competitive on the upper tail, where the baseline QRF already performed well.

In summary, v2 provides properly calibrated and regime-aware prediction intervals at the expense of some sharpness. The next step will be to determine whether these trade-offs are acceptable for your application or whether further tuning (e.g. adjusting the number of trees, minimum leaf size, or exploring different calibration stratifications) can reduce pinball loss without sacrificing coverage.

D.2.3.3 Version 3

D.2.3.4 Reflection on Tuned QRF (v3)

After hyperparameter tuning and the inclusion of an extended quantile grid, my third version of the Quantile Regression Forest has markedly improved performance. Average pinball losses for =0.05–0.95 now range from roughly 0.012 to 0.067, a dramatic reduction compared with the previous conformalized model (v2), which hovered between 0.07 and 0.15. For context, the LightGBM v4 baseline delivered pinball losses of about 0.03–0.07 across the 0.10–0.90 quantiles. In other words, the tuned QRF now outperforms LightGBM in the lower and upper tails (e.g. 0.021 at =0.10 vs. ~0.03 for LGBM) and matches it around the median (0.067 vs. ~0.066 for LGBM). This suggests that the forest, once properly calibrated and tuned, can deliver sharp, well-calibrated intervals even in highly volatile crypto markets.

The feature-importance analysis reveals that momentum and oscillator variables dominate: percentage rate of change (proc), stochastic %K (stoch_k), Bollinger band width (bollinger_b) and commodity channel index (cci) are among the top contributors. Liquidity and volume proxies, such as on-balance volume (obv) and price-volume, also rank highly, indicating that flow information carries significant predictive power for 72-hour returns. Volatility metrics (roc_3, parkinson_vol_36h, vol_std_7bar), longer-horizon returns (logret_36h), and selected on-chain variables (e.g. holder_growth_1bar/7d, tx_per_account) provide additional signal. Conversely, some engineered flags (extreme_flag1, tail_asym) and cyclical features appear to contribute little, suggesting they could be removed in later models to simplify the feature set without sacrificing performance.

Calibration pipeline. I correct the residual-quantile rule for conformal offsets by using $\delta_\tau = Q_\tau(r)$ with residuals $r = y - \hat{q}_\tau$ and map my numeric `vol_regime` quintiles to `"quiet"`, `"mid"`, and `"volatile"`. I apply these offsets to all and enforce non-crossing via isotonic regression. To achieve nominal **two-sided** coverage, I add a **split-conformal** inflation: on the calibration window I compute nonconformity scores $s = \max(q_{lo} - y,\ y - q_{hi})$, take the rank-based quantile $\delta = Q_{\lceil (n+1)\,c \rceil}(s)$ for coverage $c \{0.80,0.90\} $, and widen the test intervals by $\pm\delta$. This preserves the good per- reliability while pushing the 80%/90% bands toward nominal with minimal extra width.

Best parameters: {'n_estimators': 1467, 'max_features_choice': 'fraction', 'max_features': 0.9984302781608038, 'min_samples_leaf': 5, 'max_depth': 24} Best average pinball loss: 0.038536808768904925

What the plots show.

- **Global reliability:** =0.05 and =0.10 hug y=x (good), but **=0.25 jumps to ~0.62** and =0.50 sits ~0.74. Upper quantiles (0.75–0.95) track y=x closely.

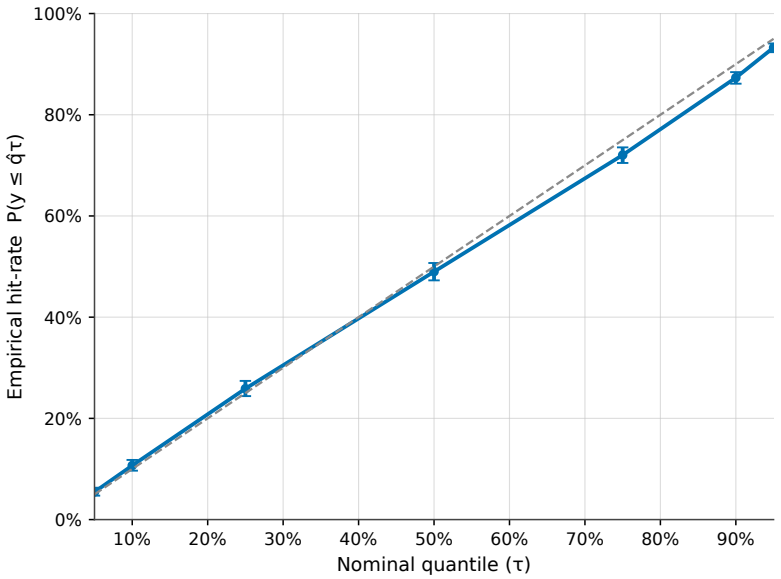


Figure D.29: QRF v3 reliability curve (global): empirical hit-rate vs nominal with Wilson CIs; ideal line shown.

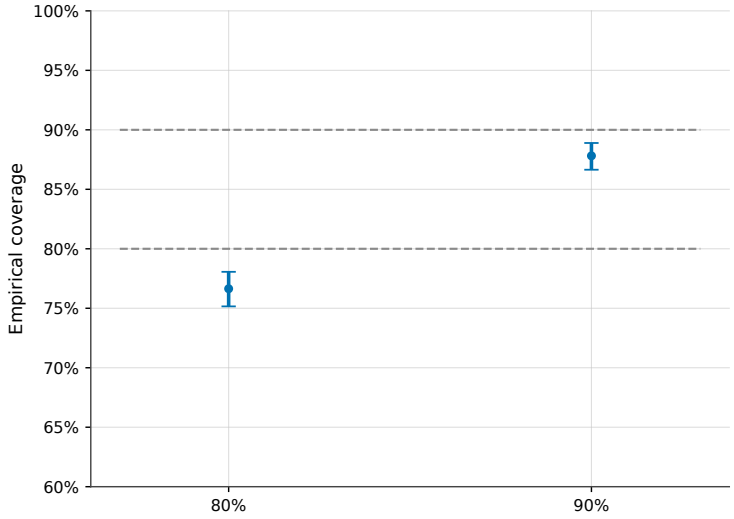


Figure D.30: QRF v3 empirical coverage of 80% and 90% intervals with Wilson CIs; dashed lines mark nominal levels.

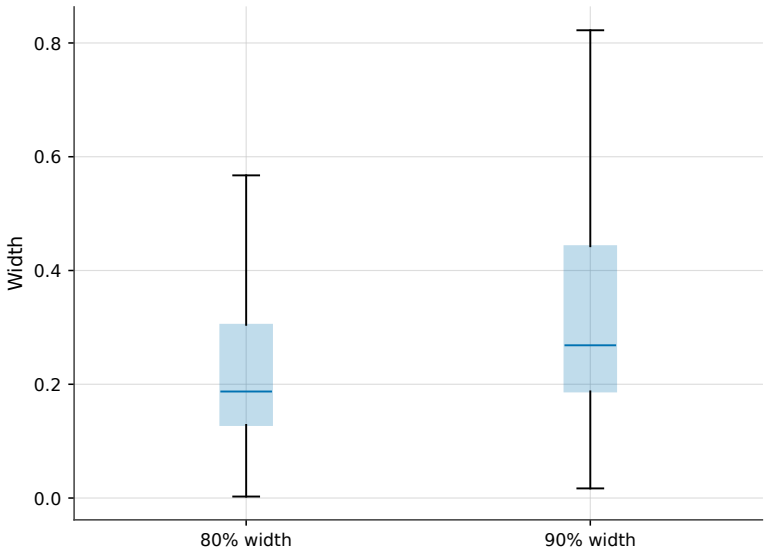


Figure D.31: QRF v3 interval width distributions for 80% and 90% prediction intervals.

- **By regime:** the **=0.25 kink persists across narrow/mid/wide** regimes, so it's systematic, not regime-specific.
- **Coverage vs nominal:** mirrors the above—slight under-coverage at 80%, closer at 90%.
- **Width distributions:** 90% bands are wider (as expected) with a long right tail during volatile periods.

D.2.3.5 Version 4 (Final)

What I changed. - After inspecting reliability curves, I identified a calibration error in my conformal shift rule for lower quantiles. I had incorrectly used $Q_{1-\tau}(r)$ instead of $Q_\tau(r)$ for residuals $r = y - \hat{q}_\tau$. I corrected the offsets to $\delta_\tau = Q_\tau(r)$ for all , keeping the regime-aware split on tails and the isotonic non-crossing step.

- I audited the volatility regime input used for regime-aware calibration. My feature table encodes vol_regime as an integer quintile in {0,1,2,3,4}, whereas my calibration code expected string labels ("quiet"/"volatile"). As a result, the quiet/volatile masks were empty and the tail offsets defaulted to global (or ~zero), i.e. regime-awareness was effectively off. I fixed this by mapping {0,1}→quiet, {3,4}→volatile, and {2}→mid, with warm-up NAs assigned to mid. I also retained a fallback that derives regimes from a past-volatility proxy (e.g., gk_vol_36h) if vol_regime is not available. **Why.**
- This ensures the adjusted quantiles satisfy $\mathbb{P}(y \le \hat{q}_\tau) \approx \tau$ uniformly across , preventing the inflated hit-rates previously observed around =0.25–0.50 and stabilising median calibration.
- The purpose of regime-aware calibration is to prevent under-coverage in turbulent periods without widening bands in calm periods. Ensuring the regime signal is recognised by the calibration step is essential; otherwise offsets can be biased toward average conditions.

D.2.4 Applications to Trading:

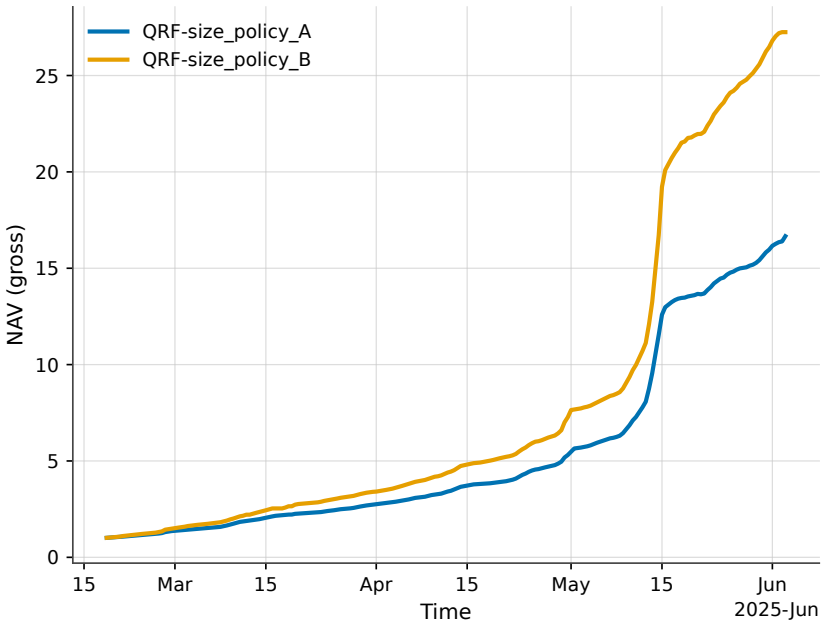


Figure D.32: Risk-aware sizing equity curves (72-hour step).

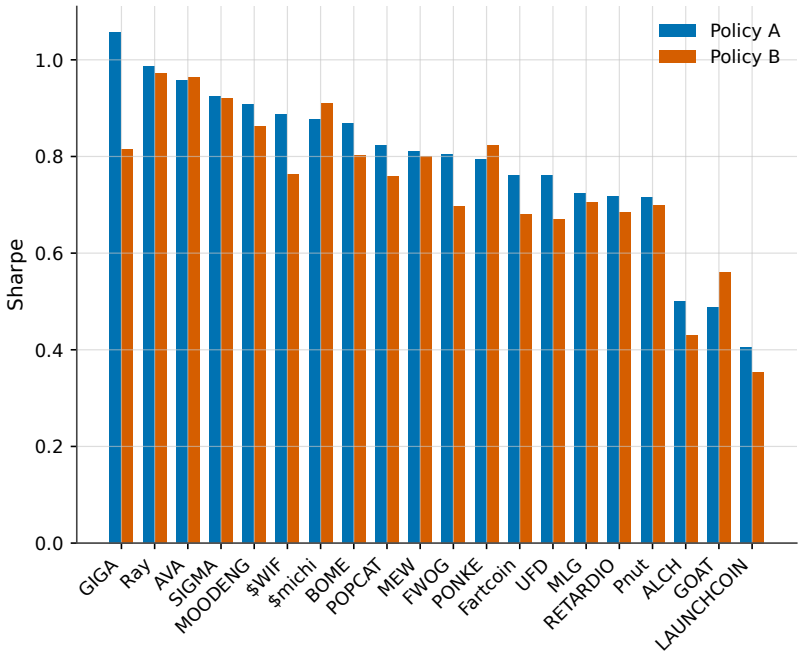


Figure D.33: Per-token Sharpe for the top 20 tokens under Policy A and Policy B.

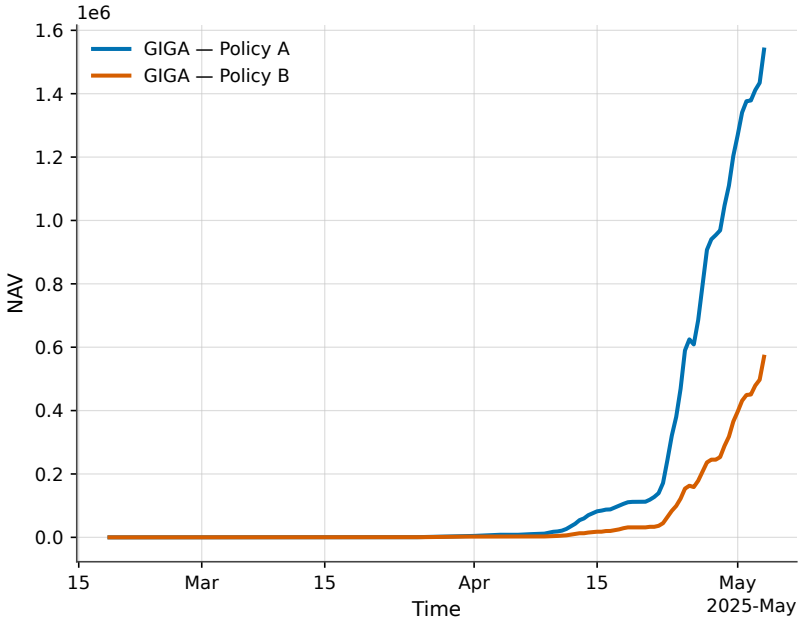


Figure D.34: Equity curve for the selected token under Policy A vs Policy B.

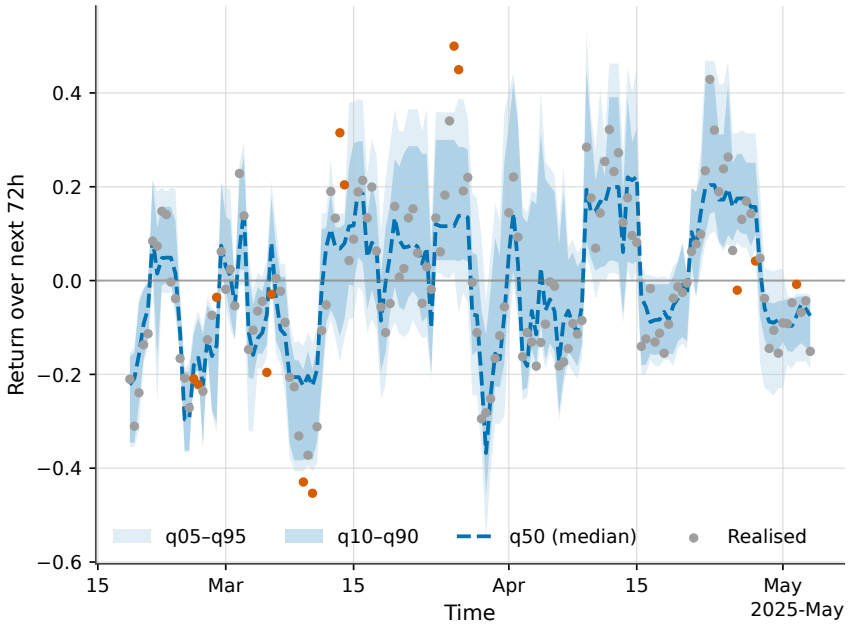


Figure D.35: Predictive fan (q05–q95, q10–q90) with median and realised 72-hour returns.

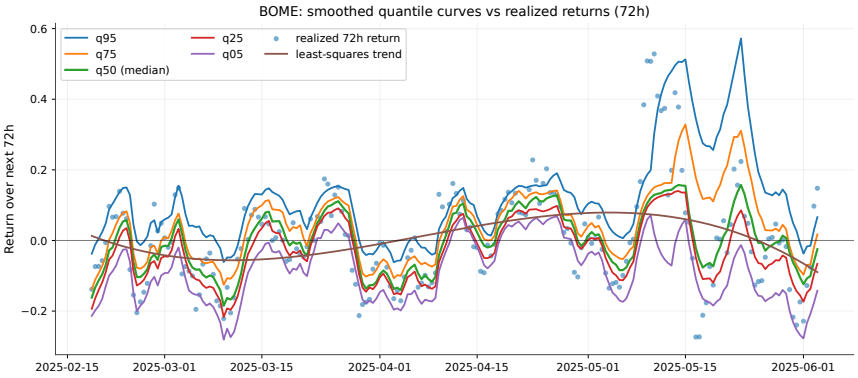


Figure D.36: Smoothed quantile "spaghetti" (q05…q95) vs realised returns and a least-squares trend.

E Appendix 5: Code

This section contain all major code from EDA, Feature Engineering, Model Building, Analysis and Testing. It will be strucutred in the chronological order of this project. This project remains open-soruce, thus all code from Data ingestion to my specific models can be found on GitHub - `https://github.com/KetchupJL/solana-qrf-interval-forecasting`.

This section won't include code from creating figures, tables, data ingestion, environment setting and essential set ups. For full notebooks with detailed write ups and documentation, visit the projects GitHub repository for access.

Due to the huge amount of code I developed during this project, large parts of the project arent included within this paper or appendix, but follow the relevent hyperlinks to access them on GitHub.

Links to Full Notebooks + Scripts

- 10+ Data Ingestion Scripts
- Data Processing Scripts
- EDA Scripts
- Feature Engineering Scripts
- Model Building, Analysis and Backtesting Scripts

E.1 Data Processing and EDA

See the full notebooks here:

- 1 Loading, Preprocessing and Creating Master Dataset
- 2 Cleaning and checking OHLCV Data
- 3 OHLCV Data Imputation
- 4 EDA Missingness
- 5 EDA Return Analysis
- 6 Correlation Analysis
- 7 Interval Calibration Analysis
- 8 CQR Rolling Calibration
- 9 EDA Report (containing all findings)

E.1.0.1 EDA

Some snippets from the EDA:

- **Creating `logret_12h` and `logret72h` variables**

```
# Compute Returns
df = df.sort_values(['token', 'timestamp'])

# 12h log return
df['logret_12h'] = df.groupby('token')['close_usd'].transform(lambda x: np.log(x / x.shift(1

# 72h log return
df['logret_72h'] = df.groupby('token')['close_usd'].transform(lambda x: np.log(x.shift(-6) /
```

- **Initial Data Summary**

```
# Schema Summary: Data Types, Uniqueness, Missingness
import pandas as pd
import numpy as np

# Shape of dataset
print("Dataset shape:", df.shape)

# Token count (assuming token column exists)
if 'token' in df.columns:
    print("Unique tokens:", df['token'].nunique())

# Time span
print("Date range:", df['timestamp'].min(), "to", df['timestamp'].max())

# Summary of dtypes, unique counts, missing values
summary = pd.DataFrame({
    'dtype': df.dtypes,
    'n_unique': df.nunique(),
    'pct_missing': df.isnull().mean() * 100
}).sort_values(by='pct_missing', ascending=False)

display(summary)
```

- **Full Feature Missingness Audit (excluding holder count)**

```
# Drop holder_count column and perform missing audit
no_holder_df = df.drop(columns=['holder_count'])

# Compute missing percentage for all remaining columns
missing_summary_alltime = pd.DataFrame({
    'dtype': no_holder_df.dtypes,
    'n_unique': no_holder_df.nunique(),
    'pct_missing': no_holder_df.isnull().mean() * 100
}).sort_values(by='pct_missing', ascending=False)
```

```
missing_summary_alltime.style.format({'pct_missing': "{:.2f}%"})
```

- **Correlation and Redundancy Analysis (spearman plot)**

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# 1. dataset
df = pd.read_parquet("C:/Users/james/OneDrive/Documents/GitHub/solana-qrf-interval-forecasti

# 2. Rename only the core features for clarity
rename_map = {
    'close_usd': 'token_close_usd',
    'volume_usd': 'token_volume_usd',
    'logret_12h': 'return_12h',
    'logret_72h': 'return_72h',
    'realized_vol_12h': 'realized_vol_12h',
    'rolling_skew_50': 'rolling_skew_50',
    'tail_asymmetry': 'tail_asymmetry',
    'extreme_freq': 'extreme_freq',
    'holder_count': 'holder_count',
    'new_token_accounts': 'new_token_accounts',
    'transfer_count': 'transfer_count',
    'btc_eth_price_btc_close': 'btc_close_usd',
    'btc_eth_price_eth_close': 'eth_close_usd',
    'sol_price_close': 'sol_close_usd',
    'sol_price_volume': 'sol_volume_usd',
    'network_tx_tx_count': 'network_tx_count',
    'tvl_tvl_usd': 'tvl_usd',
    'tvl_tvl_change_12h': 'tvl_change_12h',
    'sol_return': 'sol_return'
}
df = df.rename(columns=rename_map)

# 3. Select only these numeric features
features = [
    'token_close_usd', 'token_volume_usd',
    'return_12h', 'return_72h',
    'realized_vol_12h', 'rolling_skew_50',
    'tail_asymmetry', 'extreme_freq',
    'holder_count', 'new_token_accounts', 'transfer_count',
    'sol_close_usd', 'btc_close_usd', 'eth_close_usd',
    'tvl_usd', 'tvl_change_12h', 'network_tx_count',
    'sol_return'
]

# 4. Subset and drop any rows with missing data in these features
sub = df[features].dropna()
```

```
# 5. Compute Pearson & Spearman correlation matrices
pearson = sub.corr(method='pearson')
spearman = sub.corr(method='spearman')

# 6. Plot Pearson correlation heatmap
plt.figure()
sns.heatmap(pearson, annot=True, fmt=".2f", cmap="vlag", center=0, linewidths=0.5)
plt.title("Pearson Correlation Matrix (Key Features)")
plt.tight_layout()
plt.gcf().savefig("/Users/james/OneDrive/Documents/GitHub/solana-qrf-interval-forecasting/pa
plt.show()

# 7. Plot Spearman correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(spearman, annot=True, fmt=".2f", cmap="vlag", center=0, linewidths=0.5)
plt.title("Spearman Correlation Matrix (Key Features)")
plt.tight_layout()
plt.show()

# 8. Identify highly correlated feature pairs (|r| > 0.85)
mask = np.triu(np.ones_like(pearson, dtype=bool), k=1)
high_corr = (
    pearson.where(mask)
           .stack()
           .loc[lambda s: s.abs() > 0.85]
           .sort_values(ascending=False)
)
```

- **Interval Calibration Analysis**

```
import pandas as pd
from quantile_forest import RandomForestQuantileRegressor
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

#Load dataset
file_path = "C:/Users/james/OneDrive/Documents/GitHub/solana-qrf-interval-forecasting/data/C
df = pd.read_parquet(file_path)

#Using simple features
features = ['token_volume_usd', 'holder_count', 'sol_volume_usd', 'realized_vol_12h']
target = 'return_12h'

model_df = df.dropna(subset=features + [target])
X = model_df[features]
y = model_df[target]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

```
qrf = RandomForestQuantileRegressor(random_state=0, n_estimators=100)
qrf.fit(X_train, y_train)

alphas = np.linspace(0.5, 0.95, 10)
results = []

for alpha in alphas:
    lower_q = (1 - alpha) / 2
    upper_q = 1 - lower_q

    # when you ask for a single quantile, predict() returns 1D
    lower = qrf.predict(X_test, quantiles=[lower_q])
    upper = qrf.predict(X_test, quantiles=[upper_q])

    covered = ((y_test >= lower) & (y_test <= upper)).mean()
    width   = (upper - lower).mean()

    results.append({
        'nominal':   alpha,
        'empirical': covered,
        'width':     width
    })

qrf_cal = pd.DataFrame(results)
```

E.1.1 Feature Engineering

See the full notebooks here: - 1 Setting up the full spectrum of Financial Features - 2 Building a more robust Feature Engineering Pipeline - 3 Feature Pruning

Some snippets:

- **Creating some essential financial features**

```
def compute_base_features(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    new_cols = [
        'logret_12h', 'logret_36h', 'rsi_14', 'roc_3', 'realized_vol_36h',
        'atr_14', 'spread', 'depth', 'vol_spike', 'delta_wallets',
        'tx_count_12h', 'ret_SOL', 'ret_BTC', 'ret_ETH', 'tvl_dev'
    ]
    df.drop(columns=[c for c in new_cols if c in df.columns], inplace=True, errors='ignore')

    df['timestamp'] = pd.to_datetime(df['timestamp'])
    df.sort_values(['token', 'timestamp'], inplace=True)
    g = df.groupby('token')
    volume = df.get('token_volume_usd', df.get('volume'))

    df['logret_12h'] = g['token_close_usd'].transform(lambda x: np.log(x / x.shift(1)))
```

```
    df['logret_36h'] = g['token_close_usd'].transform(lambda x: np.log(x / x.shift(3)))
    df['rsi_14'] = g['token_close_usd'].transform(lambda x: rsi(x, 14))
    df['roc_3'] = g['token_close_usd'].transform(lambda x: (x / x.shift(3) - 1) * 100)

    df['realized_vol_36h'] = df.groupby('token')['logret_12h'].transform(lambda x: x.rolling

    df['atr_14'] = df.groupby('token', group_keys=False).apply(lambda grp: atr(grp.get('high

    if {'best_ask', 'best_bid'}.issubset(df.columns):
        mid = (df['best_ask'] + df['best_bid']) / 2
        df['spread'] = (df['best_ask'] - df['best_bid']) / mid
    else:
        df['spread'] = np.nan

    if {'bid_size', 'ask_size'}.issubset(df.columns):
        df['depth'] = df['bid_size'] + df['ask_size']
    else:
        df['depth'] = np.nan

    if volume is not None:
        df['vol_spike'] = g[volume.name].transform(lambda x: x / x.rolling(14).mean())
    else:
        df['vol_spike'] = np.nan

    uniq_wallets = df.get('unique_wallets', df.get('holder_count'))
    if uniq_wallets is not None:
        df['delta_wallets'] = g[uniq_wallets.name].transform(lambda x: x.diff())
    else:
        df['delta_wallets'] = np.nan

    df['tx_count_12h'] = df.get('tx_count', df.get('network_tx_count'))

    if 'sol_close_usd' in df.columns:
        df['ret_SOL'] = df['sol_close_usd'].pct_change() * 100
    if 'btc_close_usd' in df.columns:
        df['ret_BTC'] = df['btc_close_usd'].pct_change() * 100
    if 'eth_close_usd' in df.columns:
        df['ret_ETH'] = df['eth_close_usd'].pct_change() * 100
    if 'tvl_usd' in df.columns:
        df['tvl_dev'] = (df['tvl_usd'] / df['tvl_usd'].rolling(14).mean() - 1) * 100

    return df
```

- **Creating Tail Features**

```
def tail_features(df: pd.DataFrame) -> pd.DataFrame:
    """
    Flag extreme 12-h moves and rolling tail statistics.
    Threshold = |return_12h| > 2.5 × rolling  14
    """
    g = df.groupby("token")
```

```
    ret  = g["token_close_usd"].pct_change()
    sigma14 = ret.groupby(df["token"]).transform(lambda s: s.rolling(14).std())
    extreme = (ret.abs() > 2.5 * sigma14).astype("int")

    df["extreme_move1"]      = extreme
    df["extreme_flag1"]      = extreme
    df["tail_positive"]          = (ret >  2.5 * sigma14).astype("int")
    df["tail_negative"]          = (ret < -2.5 * sigma14).astype("int")
    # Use .astype(int) to ensure subtraction works without dtype issues
    df["tail_asym"] = df["tail_positive"].astype(int) - df["tail_negative"].astype(int)
    df["extreme_count_72h"] = extreme.groupby(df["token"]).transform(lambda s: s.rolling(6)

    return df
```

- **Pruning - checking for multicollinearity**

```
from itertools import combinations

#split Stage-1 list back into numeric vs. categorical
num_keep = [c for c in predictors_stage1 if c in num_feats]
cat_keep = [c for c in predictors_stage1 if c in cat_feats]

#compute absolute Pearson correlation on numeric part
corr = df[num_keep].corr().abs()

#scan the upper triangle; mark the *second* feature for dropping
to_drop = set()
for (col_i, col_j) in combinations(corr.columns, 2):
    if corr.loc[col_i, col_j] > 0.98:
        # keep the first occurrence, drop the second
        to_drop.add(col_j)

num_after = [c for c in num_keep if c not in to_drop]
predictors_stage2 = num_after + cat_keep

#inspect what was dropped
display(sorted(to_drop))
```

- **LightGBM for predictor importance**

```
X = df[predictors_stage2]          # predictors from Stage 2
y = df["return_72h"]

lgb_data = lgb.Dataset(
    X,
    label=y,
    categorical_feature=cat_keep,  # defined in Stage 1
    free_raw_data=False
)

params = dict(
```

```
    objective        = "quantile",
    alpha            = 0.5,          # median
    learning_rate    = 0.05,
    num_leaves       = 64,
    feature_fraction = 0.80,
    bagging_fraction = 0.80,
    seed             = 42,
    verbose          = -1,
)

gbm = lgb.train(
    params,
    lgb_data,
    num_boost_round = 400
)

gain = pd.Series(
    gbm.feature_importance(importance_type="gain"),
    index = predictors_stage2
).sort_values(ascending=False)

gain_pct = 100 * gain / gain.sum()
display(gain_pct.head(20).to_frame("gain_%").style.format({"gain_%":"{:.2f}"}))

#candidate list for Stage 4 pruning
threshold = 0.3                    # % of total gain
predictors_stage3 = gain_pct[gain_pct >= threshold].index.tolist()
```

E.2 Model Building

See the full notebooks here: - LQR V1 - LQR Final Model - LightGBM V1 - LightGBM V2 - LightGBM Final Model - QRF V1 - QRF V2 - QRF V3 and Final Model

For full documentation and notes, read through the notebooks. Below is the **raw** model building code.

E.2.1 Linear Quantile Regression

V1

```
import pandas as pd, numpy as np, statsmodels.api as sm
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import RobustScaler, OneHotEncoder
from joblib import Parallel, delayed
from pathlib import Path
import os, warnings, itertools
```

```
#0 · CONFIG
FEATURE_FILE = "features_v1.parquet"     # frozen Stage-6 dataset
TARGET       = "return_72h"
TAUS         = [0.05, 0.25, 0.50, 0.75, 0.95]
TRAIN, CAL, TEST = 120, 24, 6
MAX_ITER     = 2_000
N_JOBS       = max(os.cpu_count() - 1, 1)      # leave 1 core free
OUT_METRICS  = Path("stage7_linQR_pinball.csv")
OUT_PRED     = Path("stage7_linQR_preds.csv")

warnings.filterwarnings("ignore", category=UserWarning, module="statsmodels")

#1 · LOAD & PREP
df = pd.read_parquet(FEATURE_FILE)

EXPLICIT_CAT = ["day_of_week","extreme_flag1","momentum_bucket","tail_asym"]
for c in EXPLICIT_CAT:
    if c in df.columns:
        df[c] = df[c].astype("category")

drop = ["timestamp", "token", TARGET]
cat_feats = [c for c in df.columns if df[c].dtype.name == "category"]
num_feats = [c for c in df.columns if c not in drop + cat_feats]
predictors = num_feats + cat_feats

pre_template = ColumnTransformer([
    ("num", RobustScaler(), num_feats),
    ("cat", OneHotEncoder(drop="first",
                          sparse_output=False,
                          handle_unknown="ignore"), cat_feats)
])

MISSING_MEDIAN_COLS = [c for c in df.columns
                       if "holder" in c or "tx_per_account" in c]

def impute_median(X):
    X = X.copy()
    for col in MISSING_MEDIAN_COLS:
        if col in X:
            X[col] = X[col].fillna(X[col].median(skipna=True))
    return X.fillna(0)

#2 · ROLLING SPLITS
def rolling_indices(frame):
    idx = frame.index
    total = len(idx)
    for start in range(0, total - (TRAIN + CAL + TEST) + 1, TEST):
        tr = idx[start : start + TRAIN]
        te = idx[start + TRAIN + CAL : start + TRAIN + CAL + TEST]
        if len(te) == TEST:
            yield tr, te
```

```
#3 · ONE FOLD
def fit_fold(g, tr_idx, te_idx, tok):
    X_tr  = impute_median(g.loc[tr_idx, predictors])
    y_tr  = g.loc[tr_idx, TARGET].values
    X_te  = impute_median(g.loc[te_idx, predictors])
    y_te  = g.loc[te_idx, TARGET].values
    pre   = pre_template.fit(X_tr)
    X_trA = pre.transform(X_tr)
    X_teA = pre.transform(X_te)

    fold_res, fold_pred = [], []
    for tau in TAUS:
        mod = sm.QuantReg(y_tr,
                          sm.add_constant(X_trA, has_constant='add')
                         ).fit(q=tau, method="highs", max_iter=MAX_ITER)

        y_hat = mod.predict(sm.add_constant(X_teA, has_constant='add'))
        err   = y_te - y_hat
        pin   = np.maximum(tau*err, (tau-1)*err).mean()

        fold_res.append(dict(tau=tau, pinball=pin))
        fold_pred.extend([dict(timestamp  = g.loc[i, "timestamp"],
                               token      = tok,
                               tau        = tau,
                               y_true     = yt,
                               y_pred     = yh)
                          for i, yt, yh in zip(te_idx, y_te, y_hat)])
    return fold_res, fold_pred

#4 · PARALLEL TOKENS
def run_token(tok, g):
    token_metrics, token_preds = [], []
    for tr_idx, te_idx in rolling_indices(g):
        res, pred = fit_fold(g, tr_idx, te_idx, tok)
        token_metrics.extend(res)
        token_preds.extend(pred)
    return token_metrics, token_preds

results = Parallel(n_jobs=N_JOBS, verbose=5)(
    delayed(run_token)(tok, grp) for tok, grp in df.groupby("token")
)

#flatten
metrics = list(itertools.chain.from_iterable(r[0] for r in results))
preds   = list(itertools.chain.from_iterable(r[1] for r in results))

#5 · SAVE & REPORT
pd.DataFrame(metrics).to_csv(OUT_METRICS, index=False)
pd.DataFrame(preds  ).to_csv(OUT_PRED,   index=False)
```

LQR Final Model

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from statsmodels.regression.quantile_regression import QuantReg as QR

num_cols = [c for c in feat_cols if df[c].dtype != "object"]
cat_cols = ["token", "momentum_bucket", "day_of_week"]      # treat as categoricals

pre = ColumnTransformer([
        ("num", StandardScaler(), num_cols),
        ("cat", OneHotEncoder(drop="first", handle_unknown="ignore"), cat_cols)
      ],
      remainder="drop")

horizon       = 30    # rows in each test fold
train_window  = 120   # rows in each training window
quantiles     = [0.10, 0.50, 0.90]

fold_metrics  = []    # one row per fold × token ×
pred_records  = []    # one row per observation in every test fold

for tkn, g in df.groupby("token"):
    g = g.reset_index(drop=True)
    for f, start in enumerate(range(0,
                                    len(g) - (train_window + horizon) + 1,
                                    horizon), start=1):
        train = g.iloc[start : start + train_window]
        test  = g.iloc[start + train_window :
                       start + train_window + horizon]

        # fit scaler / encoder **only on the training slice**
        X_train = pre.fit_transform(train)
        X_test  = pre.transform(test)
        y_train = train["return_72h"].values
        y_test  = test["return_72h"].values

        fold_preds = {}
        for q in quantiles:
            model = QR(y_train, X_train).fit(q=q, max_iter=5000)
            fold_preds[q] = model.predict(X_test)

        # post-hoc non-crossing safeguard
        fold_preds[0.10] = np.minimum(fold_preds[0.10], fold_preds[0.50])
        fold_preds[0.90] = np.maximum(fold_preds[0.90], fold_preds[0.50])

        # -------- collect per-row records --------
        for i in range(len(test)):
            pred_records.append({
```

```
                "token":      tkn,
                "timestamp":  test.iloc[i]["timestamp"],
                "fold":       f,
                "y_true":     y_test[i],
                "q10_pred":   fold_preds[0.10][i],
                "q50_pred":   fold_preds[0.50][i],
                "q90_pred":   fold_preds[0.90][i],
            })

        # -------- collect per-fold metrics --------
        inside80 = ((y_test >= fold_preds[0.10]) &
                    (y_test <= fold_preds[0.90]))
        fold_metrics.append({
            "token":     tkn,
            "fold":      f,
            "tau":      "80PI",
            "coverage":  inside80.mean(),
            "width":     (fold_preds[0.90] - fold_preds[0.10]).mean(),
        })
        for q in quantiles:
            err = y_test - fold_preds[q]
            pinball = np.maximum(q*err, (q-1)*err).mean()
            fold_metrics.append({
                "token":    tkn,
                "fold":     f,
                "tau":      q,
                "pinball":  pinball
            })

#3.  Save artefacts
pd.DataFrame(pred_records).to_csv("lqr_pred_paths.csv",  index=False)
pd.DataFrame(fold_metrics).to_csv("lqr_fold_metrics.csv", index=False)

print("Finished!  Predictions → lqr_pred_paths.csv;  metrics → lqr_fold_metrics.csv")
```

E.2.2 Light GBM

V1

```
import pandas as pd, numpy as np, lightgbm as lgb
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from joblib import Parallel, delayed
import os, itertools, warnings, json

DATA_FILE = "features_v1_tail.csv"
TARGET    = "return_72h"
QUANTS    = [0.05, 0.25, 0.50, 0.75, 0.95]
```

```
df = (pd.read_csv(DATA_FILE, parse_dates=["timestamp"])
        .sort_values(["token", "timestamp"])
        .reset_index(drop=True))

cat_cols = ["day_of_week","momentum_bucket", "extreme_flag1", "tail_asym","vol_regime", "tok
num_cols = [c for c in df.columns
            if c not in cat_cols + ["timestamp", TARGET]]

#one-hot → dense matrix; LightGBM handles NaN in numeric naturally

pre = ColumnTransformer([
        ("cats", OneHotEncoder(drop="first",
                               handle_unknown="ignore",
                               sparse_output=False), cat_cols)
      ],
      remainder="passthrough")

TRAIN, CAL, TEST = 120, 24, 6     # bars (~60d, 12d, 3d)

def rolling_splits(idx):
    for start in range(0, len(idx) - (TRAIN + CAL + TEST) + 1, TEST):
        tr = idx[start : start + TRAIN]
        cal = idx[start + TRAIN : start + TRAIN + CAL]
        te = idx[start + TRAIN + CAL : start + TRAIN + CAL + TEST]
        if len(te) == TEST:
            yield tr, cal, te

def fit_one_fold(g, tr_idx, cal_idx, te_idx):
    """
    Fit LightGBM-quantile on one rolling window and return
    • fold_pred : list[dict]  (row-level predictions)
    • fold_res  : list[dict]  (fold-level pinball loss)
    """
    #   matrices
    X_tr  = pre.fit_transform(g.loc[tr_idx, cat_cols + num_cols])
    y_tr  = g.loc[tr_idx, TARGET].values
    X_cal = pre.transform(g.loc[cal_idx, cat_cols + num_cols])
    y_cal = g.loc[cal_idx, TARGET].values
    X_te  = pre.transform(g.loc[te_idx, cat_cols + num_cols])
    y_te  = g.loc[te_idx, TARGET].values

    token_id = g["token"].iloc[0]      # ← safe token label

    fold_pred, fold_res = [], []

    for tau in QUANTS:
        mdl = lgb.LGBMRegressor(
            objective="quantile", alpha=tau,
            n_estimators=500, learning_rate=0.05,
            max_depth=-1, subsample=0.9, colsample_bytree=0.9,
```

```
            min_child_samples=20, random_state=42
        )
        mdl.fit(X_tr, y_tr)

        #   base predictions
        cal_hat = mdl.predict(X_cal)
        te_hat  = mdl.predict(X_te)

        #   split-conformal adjustment
        resid = y_cal - cal_hat
        if tau < 0.5:
            adj = np.quantile(np.maximum(resid, 0), 1 - tau)
            te_adj = te_hat - adj
        elif tau > 0.5:
            adj = np.quantile(np.maximum(-resid, 0), 1 - (1 - tau))
            te_adj = te_hat + adj
        else:                       #   = 0.50
            te_adj = te_hat

        #   per-row predictions
        fold_pred.extend({
            "timestamp": g.loc[i, "timestamp"],
            "token":     token_id,
            "tau":       tau,
            "y_true":    yt,
            "y_pred":    yp
        } for i, yt, yp in zip(te_idx, y_te, te_adj))

        #   fold-level pinball loss
        err = y_te - te_adj
        pin = np.maximum(tau*err, (tau-1)*err).mean()
        fold_res.append({
            "token":   token_id,
            "tau":     tau,
            "pinball": pin
        })

    return fold_pred, fold_res

def run_token(tok, grp):
    preds, metrics = [], []
    for tr, cal, te in rolling_splits(grp.index):
        p, m = fit_one_fold(grp, tr, cal, te)
        preds.extend(p); metrics.extend(m)
    return preds, metrics

n_jobs = max(os.cpu_count()-1, 1)
results = Parallel(n_jobs=n_jobs, verbose=5)(
    delayed(run_token)(tok, grp.reset_index(drop=True))
    for tok, grp in df.groupby("token"))
```

```
preds   = list(itertools.chain.from_iterable(r[0] for r in results))
metrics = list(itertools.chain.from_iterable(r[1] for r in results))

pd.DataFrame(preds).to_csv("stage7_lgb_preds.csv", index=False)
pd.DataFrame(metrics).to_csv("stage7_lgb_pinball.csv", index=False)

print(pd.DataFrame(metrics)
        .groupby("tau")["pinball"].mean()
        .round(4))
```

V2

```
#Optuna search space


def suggest_params(trial, tau: float) -> dict:
    return {
        # ----- core CQR settings -----
        "objective" : "quantile",
        "metric"    : "quantile",
        "alpha"     : tau,
        "device_type": "gpu" if gpu_available else "cpu",

        # ----- tree complexity -----
        "learning_rate" : trial.suggest_float("lr",      0.005, 0.1,  log=True),
        "num_leaves"    : trial.suggest_int(  "leaves",      32, 256, log=True),
        "max_depth"     : trial.suggest_int(  "depth",        4, 14),
        "min_data_in_leaf":
                          trial.suggest_int(  "min_leaf",     5, 300, log=True),

        # ----- randomness & regularisation -----
        "feature_fraction": trial.suggest_float("feat_frac", 0.4, 1.0),
        "bagging_fraction": trial.suggest_float("bag_frac",  0.4, 1.0),
        "bagging_freq"    : trial.suggest_int(  "bag_freq",      0, 15),

        # **FIX**: low bound must be > 0 when log=True  → use 1e-8
        "lambda_l1" : trial.suggest_float("l1", 1e-8, 5.0, log=True),
        "lambda_l2" : trial.suggest_float("l2", 1e-8, 5.0, log=True),

        "min_gain_to_split":
                          trial.suggest_float("gamma",     0.0, 0.4),

        # ----- training length -----
        "num_iterations"        : 8000,
        "early_stopping_round"  : 400,      # (LightGBM's param without the "s")
        "verbosity"             : -1,
        "seed"                  : 42,
        "n_jobs"                : -1,       # all 24 logical threads
    }

#3.  Objective function uses the *existing* X, y, cat_idx
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_pinball_loss      # scikit-learn  1.1
import lightgbm as lgb

def pinball(y_true, y_pred, tau):
    """Lightweight pinball without sklearn if preferred."""
    diff = y_true - y_pred
    return np.maximum(tau*diff, (tau-1)*diff).mean()

def objective(trial, tau):
    params = suggest_params(trial, tau)

    X_tr, X_val, y_tr, y_val = train_test_split(
        X, y, test_size=0.15, random_state=trial.number)

    #   pass DataFrames directly
    lgb_train = lgb.Dataset(X_tr, label=y_tr)
    lgb_val   = lgb.Dataset(X_val, label=y_val, reference=lgb_train)

    booster = lgb.train(params,
                        train_set=lgb_train,
                        valid_sets=[lgb_val],
                        verbose_eval=False)

    y_hat = booster.predict(X_val, num_iteration=booster.best_iteration)
    loss  = pinball(y_val, y_hat, tau)
    trial.set_user_attr("best_iter", booster.best_iteration)
    return loss


#5.  Run Optuna - **quiet** parallel search (20 workers)
#One loop per quantile 0.05 … 0.95

import optuna, time, json
from optuna.samplers import TPESampler
from optuna.pruners  import HyperbandPruner

optuna.logging.set_verbosity(optuna.logging.WARNING)          # mute per-trial chatter

def _heartbeat(tau):
    """Print a single status line every 30 finished trials."""
    def cb(study, trial):
        if len(study.trials) % 30 == 0:
            print(f" ={tau:.2f} | {len(study.trials):3d} trials "
                  f"| best pinball = {study.best_value:.4f}")
    return cb


QUANTS       = [0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95]
best_params  = {}
```

```
for tau in QUANTS:

    sampler = TPESampler(seed=42, multivariate=True)
    pruner  = HyperbandPruner(min_resource=200, max_resource=8000)

    study = optuna.create_study(
        direction   = "minimize",
        sampler     = sampler,
        pruner      = pruner,
        study_name  = f"lgb_cqr_tau{tau:.2f}",
        storage     = f"sqlite:///lgb_cqr_tau{tau:.2f}.db",
        load_if_exists=True
    )

    t0 = time.time()
    study.optimize(
        lambda t: objective(t, tau),
        n_trials   = 300,
        n_jobs     = 20,              # 24-core workstation - leave 4 for OS/Jupyter
        timeout    = 3 * 3600,
        callbacks  = [_heartbeat(tau)],
        show_progress_bar = False
    )

    print(f"   ={tau:.2f}: best pinball = {study.best_value:.4f} "
          f"@ {study.best_trial.user_attrs['best_iter']} trees "
          f"({time.time()-t0:.1f}s)")

    p = study.best_params
    p.update(objective="quantile",
             metric   ="quantile",
             alpha    = tau,
             num_iterations = study.best_trial.user_attrs["best_iter"])
    best_params[tau] = p

json.dump(best_params, open("best_lgb_cqr_params.json", "w"), indent=2)



best_params = json.load(open("best_lgb_cqr_params.json"))

QUANTS    = [0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95]

df = (pd.read_csv(DATA_FILE, parse_dates=["timestamp"])
        .sort_values(["token", "timestamp"])
        .reset_index(drop=True))

cat_cols = ["day_of_week","momentum_bucket", "extreme_flag1", "tail_asym","vol_regime", "tok
num_cols = [c for c in df.columns
            if c not in cat_cols + ["timestamp", TARGET]]
```

```
#one-hot → dense matrix; LightGBM handles NaN in numeric naturally
pre = ColumnTransformer([
        ("cats", OneHotEncoder(drop="first",
                               handle_unknown="ignore",
                               sparse_output=False), cat_cols)
      ],
      remainder="passthrough")

TRAIN, CAL, TEST = 120, 24, 6      #   60 d · 12 d · 3 d

def rolling_splits(idx):
    for start in range(0, len(idx) - (TRAIN+CAL+TEST) + 1, TEST):
        tr  = idx[start : start+TRAIN]
        cal = idx[start+TRAIN : start+TRAIN+CAL]
        te  = idx[start+TRAIN+CAL : start+TRAIN+CAL+TEST]
        if len(te) == TEST:
            yield tr, cal, te

def fit_one_fold_tuned(g, tr_idx, cal_idx, te_idx):
    X_tr  = pre.fit_transform(g.loc[tr_idx, cat_cols+num_cols])
    y_tr  = g.loc[tr_idx, TARGET].values
    X_cal = pre.transform(g.loc[cal_idx, cat_cols+num_cols])
    y_cal = g.loc[cal_idx, TARGET].values
    X_te  = pre.transform(g.loc[te_idx,  cat_cols+num_cols])
    y_te  = g.loc[te_idx,  TARGET].values

    token_id = g["token"].iloc[0]

    fold_pred, fold_res = [], []

    for tau in QUANTS:
        #-------- instantiate model with its own tuned dict --------
        p = best_params[str(tau)].copy()
        mdl = lgb.LGBMRegressor(**p)           #LGBM wrapper lets us keep .predict API
        mdl.fit(X_tr, y_tr, verbose=False)

        #-------- base preds --------
        cal_hat = mdl.predict(X_cal)
        te_hat  = mdl.predict(X_te)

        #-------- conformal adjust (same logic) --------
        resid = y_cal - cal_hat
        if tau < 0.5:
            adj    = np.quantile(np.maximum(resid, 0), 1 - tau)
            te_adj = te_hat - adj
        elif tau > 0.5:
            adj    = np.quantile(np.maximum(-resid, 0), 1 - (1 - tau))
            te_adj = te_hat + adj
        else:
            te_adj = te_hat
```

```
        #-------- per-row store --------
        fold_pred.extend({
            "timestamp": g.loc[i, "timestamp"],
            "token":     token_id,
            "tau":       tau,
            "y_true":    yt,
            "y_pred":    yp
        } for i, yt, yp in zip(te_idx, y_te, te_adj))

        #-------- fold pinball --------
        err = y_te - te_adj
        pin = np.maximum(tau*err, (tau-1)*err).mean()
        fold_res.append({"token": token_id, "tau": tau, "pinball": pin})

        del mdl; gc.collect()

    return fold_pred, fold_res

def run_token(tok, grp):
    preds, mets = [], []
    for tr, cal, te in rolling_splits(grp.index):
        p, m = fit_one_fold_tuned(grp, tr, cal, te)
        preds.extend(p);  mets.extend(m)
    return preds, mets

n_jobs = max(os.cpu_count()-1, 1)
results = Parallel(n_jobs=n_jobs, verbose=5)(
    delayed(run_token)(tok, g.reset_index(drop=True))
    for tok, g in df.groupby("token")
)

preds   = list(itertools.chain.from_iterable(r[0] for r in results))
metrics = list(itertools.chain.from_iterable(r[1] for r in results))

pd.DataFrame(preds  ).to_csv("lgb_tuned_preds.csv",   index=False)
pd.DataFrame(metrics).to_csv("lgb_tuned_pinball.csv", index=False)


print(pd.DataFrame(metrics).groupby("tau")["pinball"].mean().round(4))
```

LightGBM Final Model

```
#LightGBM-Conformal v4  -  "tighter fan" edition  (WIDE CSV OUTPUT)

import json, gc, os, warnings, itertools
import numpy  as np
import pandas as pd
import lightgbm as lgb
from joblib import Parallel, delayed
from tqdm.auto import tqdm
```

```
warnings.filterwarnings("ignore", category=UserWarning)

#0. I/O & meta
DATA_FILE   = "features_v1_tail.csv"          # cleaned matrix
PARAM_FILE  = "best_lgb_cqr_params.json"      # Optuna winners (v2)
TARGET      = "return_72h"

COVER       = 0.80         # desired PI coverage
alpha_tail  = (1 - COVER) / 2.0   # 0.10  for two-sided 80 %

QUANTS      = [alpha_tail, 0.10, 0.25, 0.50, 0.75, 0.90, 1 - alpha_tail]  # 0.10 0.25 0.5 …

#1. data
df = (pd.read_csv(DATA_FILE, parse_dates=["timestamp"])
        .sort_values(["token", "timestamp"])
        .reset_index(drop=True))

cat_cols = ["token", "momentum_bucket", "day_of_week"]
num_cols = [c for c in df.columns if c not in cat_cols + ["timestamp", TARGET]]

#cast categoricals → category dtype (LightGBM native)
for c in cat_cols:
    df[c] = df[c].astype("category")

from sklearn.preprocessing import StandardScaler, OrdinalEncoder
from sklearn.compose import ColumnTransformer

#Preprocessing pipeline: scale numerics, encode categoricals
pre = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), num_cols),
        ("cat", OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-1), cat_co
    ],
    remainder="drop"
)
pre.fit(df[cat_cols + num_cols])

#2. rolling splits -----------------
TRAIN, CAL, TEST = 120, 24, 6          # 60 d · 12 d · 3 d

def rolling_splits(idx):
    step = TEST
    for start in range(0, len(idx) - (TRAIN+CAL+TEST) + 1, step):
        tr  = idx[start : start+TRAIN]
        cal = idx[start+TRAIN : start+TRAIN+CAL]
        te  = idx[start+TRAIN+CAL : start+TRAIN+CAL+TEST]
        if len(te) == TEST:
            yield tr, cal, te

#3. conformal helper ---------------
```

```
#Load best_params from JSON file
with open(PARAM_FILE, "r") as f:
    best_params = json.load(f)

def cqr_adjust(pred_te, resid_cal, tau):
    """
    • For lower bound (tau < 0.5):  subtract q̂_{1- /2}(r )
    • For upper bound (tau > 0.5):  add     q̂_{1- /2}(r )
    Ensures   (1- ) two-sided coverage.
    """
    if tau < 0.5:
        r_plus = np.maximum(resid_cal, 0.0)
        q_adj  = np.quantile(r_plus, 1 - alpha_tail)
        return pred_te - q_adj
    elif tau > 0.5:
        r_plus = np.maximum(-resid_cal, 0.0)
        q_adj  = np.quantile(r_plus, 1 - alpha_tail)
        return pred_te + q_adj
    else:
        return pred_te          # median - no shift

def params_for_tau(tau: float) -> dict:
    # try all reasonable key variants
    for k in (tau, str(tau), f"{tau:.2f}", f"{tau:.3f}"):
        if k in best_params:
            p = best_params[k].copy()
            break
    else:                                  # no exact key found
        nearest = min(best_params.keys(),
                      key=lambda k: abs(float(k) - tau))
        p = best_params[nearest].copy()

    p["alpha"] = tau                       # overwrite with the true
    return p

#4. per-fold fit -------------------
def find_lambda(lower, upper, y_cal, cover=0.80):
    """
    Smallest non-negative   such that
        P( y   [lower-  , upper+ ] )   cover
    on the calibration slice.
    """
     = 0.0
    step = np.percentile(upper - lower, 75) * 0.02      # 2 % IQR heuristic
    while True:
        inside = ((y_cal >= (lower -  )) & (y_cal <= (upper +  ))).mean()
        if inside >= cover or   > 10.0:
            return
          += step

#fit one rolling window -----------------
```

```
def fit_one_fold(g, tr_idx, cal_idx, te_idx):
    X_tr  = pre.fit_transform(g.loc[tr_idx, cat_cols+num_cols]).astype("float32")
    y_tr  = g.loc[tr_idx, TARGET].values
    X_cal = pre.transform   (g.loc[cal_idx, cat_cols+num_cols]).astype("float32")
    y_cal = g.loc[cal_idx, TARGET].values
    X_te  = pre.transform   (g.loc[te_idx,  cat_cols+num_cols]).astype("float32")
    y_te  = g.loc[te_idx,  TARGET].values

    token_id  = g["token"].iloc[0]
    fold_pred, fold_res = [], []

    base_models, base_preds_cal, base_preds_te = {}, {}, {}
    for tau in [0.05, 0.10, 0.50, 0.90, 0.95]:
        p = params_for_tau(tau)
        p.update(num_iterations=4000, early_stopping_round=200, verbose=-1)
        mdl = lgb.LGBMRegressor(**p)
        mdl.fit(X_tr, y_tr, eval_set=[(X_cal, y_cal)], eval_metric="quantile")
        base_models[tau]      = mdl
        base_preds_cal[tau]   = mdl.predict(X_cal)
        base_preds_te[tau]    = mdl.predict(X_te)

    #Conformal adjustment for extreme quantiles (0.05, 0.95)
    adjusted_te, adjusted_cal = {}, {}
    for tau in [0.05, 0.10, 0.50, 0.90, 0.95]:
        resid_cal            = y_cal - base_preds_cal[tau]
        adjusted_te[tau]     = cqr_adjust(base_preds_te[tau], resid_cal, tau)
        adjusted_cal[tau]    = cqr_adjust(base_preds_cal[tau], resid_cal, tau)

    #Adaptive  to ensure central 80 % coverage
    lower_cal = adjusted_cal[0.10]
    upper_cal = adjusted_cal[0.90]
     _star    = find_lambda(lower_cal, upper_cal, y_cal, cover=COVER)

    #Min-width floor (15 % of  _cal)
    sigma_cal = np.std(y_cal)
    min_w     = 0.15 * sigma_cal
     _final   = np.maximum( _star, min_w)

    #Adjusted TEST predictions
    lower_te  = adjusted_te[0.10] -  _final
    upper_te  = adjusted_te[0.90] +  _final
    median_te = adjusted_te[0.50]


    #Store row-level preds (LONG) & pinball
    mapping = {
        0.05: adjusted_te[0.05],
        0.10: lower_te,
        0.25: 0.25 * lower_te + 0.75 * median_te,
        0.50: median_te,
        0.75: 0.75 * median_te + 0.25 * upper_te,
```

```
        0.90: upper_te,
        0.95: adjusted_te[0.95],
    }

    for tau, preds in mapping.items():
        # per-row predictions
        fold_pred.extend({
            "timestamp": g.loc[i, "timestamp"],
            "token":     token_id,
            "tau":       tau,
            "y_true":    yt,
            "y_pred":    yp
        } for i, yt, yp in zip(te_idx, y_te, preds))

        # pinball
        err = y_te - preds
        pin = np.maximum(tau*err, (tau-1)*err).mean()
        fold_res.append({"token": token_id, "tau": tau, "pinball": pin})

    del base_models; gc.collect()
    return fold_pred, fold_res

#5. parallel run -------------------
def run_token(tok, grp):
    preds, mets = [], []
    for tr, cal, te in rolling_splits(grp.index):
        p, m = fit_one_fold(grp, tr, cal, te)
        preds.extend(p); mets.extend(m)
    return preds, mets

results = Parallel(n_jobs=max(os.cpu_count()-2, 1), verbose=5)(
    delayed(run_token)(tok, g.reset_index(drop=True))
    for tok, g in tqdm(df.groupby("token"), desc="tokens"))

preds   = list(itertools.chain.from_iterable(r[0] for r in results))
metrics = list(itertools.chain.from_iterable(r[1] for r in results))

#6. save (WIDE preds + pinball) ----
preds_long = pd.DataFrame(preds)

#pivot to wide: one row per token/timestamp, qXX_pred columns
wide = (preds_long
        .pivot_table(index=["token","timestamp"],
                     columns="tau", values="y_pred", aggfunc="first")
        .reset_index())

tau_to_col = {0.05:"q05_pred", 0.10:"q10_pred", 0.25:"q25_pred",
              0.50:"q50_pred", 0.75:"q75_pred", 0.90:"q90_pred", 0.95:"q95_pred"}
wide = wide.rename(columns=tau_to_col)

#attach y_true (first per token/timestamp)
```

```
y_first = (preds_long.groupby(["token","timestamp"])["y_true"]
                     .first()
                     .reset_index())
wide = y_first.merge(wide, on=["token","timestamp"], how="left")

#reorder columns
ordered_cols = ["token","timestamp","y_true",
                "q05_pred","q10_pred","q25_pred","q50_pred",
                "q75_pred","q90_pred","q95_pred"]
wide = wide.reindex(columns=ordered_cols)
#
wide.to_csv("lgb_extended_preds.csv", index=False)
pd.DataFrame(metrics).to_csv("lgb_v4_pinball.csv", index=False)

#7. quick summary ------------------
met = (pd.DataFrame(metrics)
         .groupby("tau")["pinball"].mean()
         .round(4))

print(met)

#Empirical 80 % central coverage using wide preds

inside = ((wide["y_true"] >= wide["q10_pred"]) &
          (wide["y_true"] <= wide["q90_pred"])).mean()
print(f"Empirical {int(COVER*100)} % coverage : {inside*100:.2f} %")
```

E.2.3 Quantile Regression Forests

V1

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_pinball_loss
from tqdm.auto import tqdm
from quantile_forest import RandomForestQuantileRegressor
import os

#Quantiles to predict

quantiles = [0.10, 0.25, 0.50, 0.75, 0.90]


def enforce_monotonicity(q_preds: np.ndarray) -> np.ndarray:
    #Sort each row of predicted quantiles to prevent crossing
```

```
    return np.sort(q_preds, axis=1)

feature_path = "features_v1_tail.csv"
if not os.path.exists(feature_path):
    raise FileNotFoundError(f'{feature_path} not found. Place it in the working directory.')

full_df = pd.read_csv(feature_path)
full_df = full_df.sort_values(['token', 'timestamp']).reset_index(drop=True)

target_col = 'return_72h'
feature_cols = [c for c in full_df.columns if c not in [target_col, 'timestamp']]

categorical_cols = [c for c in feature_cols if c in ['token', 'momentum_bucket', 'day_of_wee
numeric_cols = [c for c in feature_cols if c not in categorical_cols]

preprocessor = ColumnTransformer([
    ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols),
    ('num', 'passthrough', numeric_cols)
])

all_preds = []
fold_losses = []

TRAIN_BARS = 120
CAL_BARS = 24
TEST_BARS = 6
STEP = 6

tokens = full_df['token'].unique().tolist()

for token in tqdm(tokens, desc='Tokens'):
    df_tok = full_df[full_df['token'] == token].reset_index(drop=True)
    n_rows = len(df_tok)
    start_idx = 0
    fold_num = 0
    while start_idx + TRAIN_BARS + CAL_BARS + TEST_BARS <= n_rows:
        train_slice = slice(start_idx, start_idx + TRAIN_BARS)
        test_slice = slice(start_idx + TRAIN_BARS + CAL_BARS, start_idx + TRAIN_BARS + CAL_

        train_df = df_tok.iloc[train_slice]
        test_df = df_tok.iloc[test_slice]

        X_train, y_train = train_df[feature_cols], train_df[target_col]
        X_test, y_test = test_df[feature_cols], test_df[target_col]

        qrf = RandomForestQuantileRegressor(n_estimators=1000,
                                            max_depth=None,
                                            min_samples_leaf=10,
                                            max_features='sqrt',
                                            random_state=42,
                                            n_jobs=-1)
```

```
        pipeline = Pipeline([
            ('preprocess', preprocessor),
            ('model', qrf)
        ])

        pipeline.fit(X_train, y_train)
        preds = pipeline.predict(X_test, quantiles=quantiles)
        preds = enforce_monotonicity(preds)

        losses = [mean_pinball_loss(y_test, preds[:, i], alpha=q) for i, q in enumerate(quan

        for idx, true_val in enumerate(y_test):
            row_pred = {'token': token, 'timestamp': test_df.iloc[idx]['timestamp'], 'y_true
            for i, q in enumerate(quantiles):
                row_pred[f'pred_q{int(q*100):02d}'] = preds[idx, i]
            all_preds.append(row_pred)

        fold_losses.append({
            'token': token,
            'fold': fold_num,
            **{f'pinball_q{int(q*100):02d}': loss for q, loss in zip(quantiles, losses)}
        })

        fold_num += 1
        start_idx += STEP
```

V2

```
def compute_decay_weights(n: int, half_life: float = 60.0) -> np.ndarray:

    #Compute exponentially decaying weights for a sequence of length n.
    #Each successive element receives weight exp(-k / (half_life / log(2))), where k is the

    decay_constant = half_life / np.log(2)
    indices = np.arange(n)[::-1]  # reverse so most recent observation has index 0
    weights = np.exp(-indices / decay_constant)
    return weights / weights.sum()


def winsorize_residuals(residuals: np.ndarray) -> np.ndarray:

    #Winsorize residuals to median ± 5 * IQR.

    if residuals.size == 0:
        return residuals
    med = np.median(residuals)
    width = iqr(residuals)
    lower = med - 5 * width
    upper = med + 5 * width
    return np.clip(residuals, lower, upper)
```

```
def isotonic_non_crossing(preds: np.ndarray, quantiles: list) -> np.ndarray:

    #Enforce monotonicity of quantile predictions using isotonic regression on each row.

    iso_preds = np.empty_like(preds)
    ir = IsotonicRegression(increasing=True, out_of_bounds='clip')
    for i in range(preds.shape[0]):
        iso_preds[i, :] = ir.fit_transform(quantiles, preds[i, :])
    return iso_preds


#Rolling parameters

train_len = 120
cal_len = 24
test_len = 6
step = 6

quantiles = [0.10, 0.25, 0.50, 0.75, 0.90]

#Placeholders for predictions and pinball losses

pred_records = []
pinball_records = []

#Loop over each token

for token in df['token'].unique():
    df_tok = df[df['token'] == token].reset_index(drop=True)
    n = len(df_tok)

    #Compute indices for rolling windows

    start = 0
    fold_idx = 0
    while start + train_len + cal_len + test_len <= n:
        train_slice = slice(start, start + train_len)
        cal_slice = slice(start + train_len, start + train_len + cal_len)
        test_slice = slice(start + train_len + cal_len, start + train_len + cal_len + test_l

        df_train = df_tok.iloc[train_slice]
        df_cal = df_tok.iloc[cal_slice]
        df_test = df_tok.iloc[test_slice]

        X_train = df_train[feature_cols]
        y_train = df_train[target_col]
        X_cal = df_cal[feature_cols]
        y_cal = df_cal[target_col]
        X_test = df_test[feature_cols]
```

```
y_test = df_test[target_col]

#Compute sample weights with exponential decay

weights = compute_decay_weights(len(y_train), half_life=60)

#Fit preprocessing and QRF model

model = RandomForestQuantileRegressor(
    n_estimators=1000,
    min_samples_leaf=10,
    max_features='sqrt',
    bootstrap=True,
    random_state=42,
    n_jobs=-1
)

#Create a pipeline so that preprocessor is fitted jointly with the model

pipe = Pipeline([
    ('preprocess', preprocessor),
    ('qrf', model)
])

#Fit

pipe.fit(X_train, y_train, qrf__sample_weight=weights)

#Predict (pass quantiles at predict-time)

preds_cal  = np.array(pipe.predict(X_cal,  quantiles=quantiles))
preds_test = np.array(pipe.predict(X_test, quantiles=quantiles))


#Compute residuals on calibration: residual = y_true - y_pred

residuals = y_cal.values.reshape(-1, 1) - preds_cal

#Mask heavy missingness rows for calibration offset estimation
if len(imputation_mask_cols) > 0:
    imputed_counts = df_cal[imputation_mask_cols].sum(axis=1)
    valid_mask = imputed_counts / len(imputation_mask_cols) < 0.3
else:
    valid_mask = np.ones(len(df_cal), dtype=bool)

#Determine volatility regime for each calibration row
regime_cal = df_cal['vol_regime'].astype(str).values

#Compute median residual for bias correction (only on valid rows)
median_bias = np.median(residuals[valid_mask, quantiles.index(0.50)])
```

```
#Initialize offset array for each quantile
offsets = np.zeros(len(quantiles))

#For each quantile compute regime-specific offset
for qi, tau in enumerate(quantiles):
    res_q = residuals[valid_mask, qi]
    res_q = winsorize_residuals(res_q)

    if tau in [0.10, 0.90] and 'volatile' in set(regime_cal):
        #mask for quiet vs volatile
        quiet_mask = (regime_cal == 'quiet') & valid_mask
        vol_mask = (regime_cal == 'volatile') & valid_mask
        if tau < 0.50:
            #lower quantile uses (1 - tau) quantile of residuals
            if res_q[quiet_mask].size > 0:
                quiet_offset = np.quantile(res_q[quiet_mask], 1 - tau)
            else:
                quiet_offset = np.quantile(res_q, 1 - tau)
            if res_q[vol_mask].size > 0:
                vol_offset = np.quantile(res_q[vol_mask], 1 - tau)
            else:
                vol_offset = quiet_offset
            count_quiet = quiet_mask.sum()
            count_vol = vol_mask.sum()
            if count_quiet + count_vol > 0:
                offsets[qi] = (count_quiet * quiet_offset + count_vol * vol_offset)
            else:
                offsets[qi] = np.quantile(res_q, 1 - tau)
        else:
            #upper quantile uses tau quantile of residuals
            if res_q[quiet_mask].size > 0:
                quiet_offset = np.quantile(res_q[quiet_mask], tau)
            else:
                quiet_offset = np.quantile(res_q, tau)
            if res_q[vol_mask].size > 0:
                vol_offset = np.quantile(res_q[vol_mask], tau)
            else:
                vol_offset = quiet_offset
            count_quiet = quiet_mask.sum()
            count_vol = vol_mask.sum()
            if count_quiet + count_vol > 0:
                offsets[qi] = (count_quiet * quiet_offset + count_vol * vol_offset)
            else:
                offsets[qi] = np.quantile(res_q, tau)
    else:
        #Non-regime specific offset
        if tau < 0.50:
            offsets[qi] = np.quantile(res_q, 1 - tau)
        elif tau > 0.50:
            offsets[qi] = np.quantile(res_q, tau)
        else:
```

```
                offsets[qi] = 0.0

        #Adjust test predictions using offsets and median bias
        adjusted_test = preds_test + offsets  # broadcast offsets across rows
        #Median bias correction
        adjusted_test[:, quantiles.index(0.50)] += median_bias

        #Enforce non-crossing
        adjusted_test = isotonic_non_crossing(adjusted_test, quantiles)

        #Evaluate pinball loss for each quantile on the test set
        for qi, tau in enumerate(quantiles):
            loss = mean_pinball_loss(y_test, adjusted_test[:, qi], alpha=tau)
            pinball_records.append({
                'token': token,
                'fold': fold_idx,
                'tau': tau,
                'pinball_loss': loss
            })

        #Save row-level predictions
        for i, row in df_test.iterrows():
            record = {
                'token': token,
                'timestamp': row['timestamp'],
                'fold': fold_idx,
                'y_true': row[target_col]
            }
            for qi, tau in enumerate(quantiles):
                record[f'q{int(tau*100)}'] = adjusted_test[i - test_slice.start, qi]
            pred_records.append(record)

        #Move window forward
        start += step
        fold_idx += 1

#Convert records to dataframes
pred_df = pd.DataFrame(pred_records)
pinball_df = pd.DataFrame(pinball_records)

#Aggregate pinball loss across folds
avg_pinball = pinball_df.groupby('tau')['pinball_loss'].mean().reset_index()
avg_pinball.rename(columns={'pinball_loss': 'avg_pinball_loss'}, inplace=True)

#Save outputs to CSV
pred_df.to_csv('qrf_v2_preds.csv', index=False)
pinball_df.to_csv('qrf_v2_pinball.csv', index=False)
avg_pinball.to_csv('qrf_v2_avg_pinball.csv', index=False)

avg_pinball
```

V3

```
#Imports
import pandas as pd
import numpy as np
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from quantile_forest import RandomForestQuantileRegressor
from sklearn.isotonic import IsotonicRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_pinball_loss
from scipy.stats import iqr
import optuna
import warnings
warnings.filterwarnings('ignore')


def compute_decay_weights(n: int, half_life: float = 60.0) -> np.ndarray:
    decay_constant = half_life / np.log(2)
    indices = np.arange(n)[::-1]
    weights = np.exp(-indices / decay_constant)
    return weights / weights.sum()


def winsorize_residuals(residuals: np.ndarray) -> np.ndarray:
    if residuals.size == 0:
        return residuals
    med = np.median(residuals)
    width = iqr(residuals)
    lower = med - 5 * width
    upper = med + 5 * width
    return np.clip(residuals, lower, upper)


def isotonic_non_crossing(preds: np.ndarray, quantiles: list) -> np.ndarray:
    iso_preds = np.empty_like(preds)
    ir = IsotonicRegression(increasing=True, out_of_bounds='clip')
    for i in range(preds.shape[0]):
        iso_preds[i, :] = ir.fit_transform(quantiles, preds[i, :])
    return iso_preds

#helpers: regime labels
def resolve_regime_labels(df_fold):
    """
    Returns string labels in {'quiet','mid','volatile'} based on df_fold['vol_regime'] if pr
    otherwise derives regimes from a volatility proxy (no look-ahead).
    """
    import numpy as np, pandas as pd

    if "vol_regime" in df_fold.columns:
        reg = df_fold["vol_regime"]
```

```
        if pd.api.types.is_numeric_dtype(reg):
            # 5-bin code → 3 regimes: 0-1 quiet, 2 mid, 3-4 volatile
            out = pd.Series(np.where(reg >= 3, "volatile",
                             np.where(reg <= 1, "quiet", "mid")),
                            index=reg.index, dtype="object")
            out[reg.isna()] = "mid"  # neutralise warm-up
            return out
        else:
            # normalise strings if they already exist
            m = {"low":"quiet","quiet":"quiet","calm":"quiet",
                 "mid":"mid","normal":"mid",
                 "high":"volatile","volatile":"volatile","wild":"volatile"}
            return reg.astype(str).str.lower().map(m).fillna("mid")

    #Fallback (if vol_regime column absent): use a past-vol proxy available at t
    proxy_candidates = [c for c in ["gk_vol_36h","parkinson_vol_36h","vol_std_7bar","downsid
    if proxy_candidates:
        v = df_fold[proxy_candidates[0]]
        q1, q2 = v.quantile([0.33, 0.66])
        out = pd.Series(np.where(v >= q2, "volatile", np.where(v <= q1, "quiet", "mid")),
                        index=v.index, dtype="object")
        out[v.isna()] = "mid"
        return out

    #Last resort: everything mid
    return pd.Series(["mid"] * len(df_fold), index=df_fold.index, dtype="object")


#Rolling settings
train_len = 120
cal_len = 24
test_len = 6
step = 6

quantiles = [0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95]

tokens_to_use = df['token'].unique()


def objective(trial: optuna.Trial) -> float:
    #Hyperparameters to tune
    n_estimators = trial.suggest_int('n_estimators', 600, 2000)
    max_features_choice = trial.suggest_categorical('max_features_choice', ['sqrt', 'log2',
    if max_features_choice == 'fraction':
        max_features = trial.suggest_float('max_features', 0.3, 1.0)
    else:
        max_features = max_features_choice
    min_samples_leaf = trial.suggest_int('min_samples_leaf', 5, 60)
    max_depth = trial.suggest_categorical('max_depth', [None] + list(range(6, 29, 2)))

    total_loss = []
```

```
for token in tokens_to_use:
    df_tok = df[df['token'] == token].reset_index(drop=True)
    n = len(df_tok)
    start = 0
    fold_count = 0
    while start + train_len + cal_len + test_len <= n:
        if fold_count >= 10:
            break
        train_slice = slice(start, start + train_len)
        cal_slice = slice(start + train_len, start + train_len + cal_len)
        test_slice = slice(start + train_len + cal_len, start + train_len + cal_len + te

        df_train = df_tok.iloc[train_slice]
        df_cal = df_tok.iloc[cal_slice]
        df_test = df_tok.iloc[test_slice]

        X_train = df_train[feature_cols]
        y_train = df_train[target_col]
        X_cal = df_cal[feature_cols]
        y_cal = df_cal[target_col]
        X_test = df_test[feature_cols]
        y_test = df_test[target_col]

        weights = compute_decay_weights(len(y_train), half_life=60)

        model = RandomForestQuantileRegressor(
            n_estimators=n_estimators,
            min_samples_leaf=min_samples_leaf,
            max_features=max_features,
            max_depth=max_depth,
            bootstrap=True,
            random_state=42,
            n_jobs=-1
        )

        pipe = Pipeline([
            ('preprocess', preprocessor),
            ('qrf', model)
        ])

        #Fit pipeline: only pass sample weights to qrf
        pipe.fit(X_train, y_train, qrf__sample_weight=weights)

        #Predict quantiles
        preds_cal = np.array(pipe.predict(X_cal, quantiles=quantiles))
        preds_test = np.array(pipe.predict(X_test, quantiles=quantiles))

        residuals = y_cal.values.reshape(-1, 1) - preds_cal

        if len(imputation_mask_cols) > 0:
```

```
    imputed_counts = df_cal[imputation_mask_cols].sum(axis=1)
    valid_mask = imputed_counts / len(imputation_mask_cols) < 0.3
else:
    valid_mask = np.ones(len(df_cal), dtype=bool)

regime_cal = df_cal['vol_regime'].astype(str).values
offsets = np.zeros(len(quantiles))
median_bias = np.median(residuals[valid_mask, quantiles.index(0.50)])

for qi, tau in enumerate(quantiles):
    res_q = winsorize_residuals(residuals[valid_mask, qi])
    if tau in [0.05, 0.10, 0.90, 0.95]:
        quiet_mask = (regime_cal == 'quiet') & valid_mask
        vol_mask = (regime_cal == 'volatile') & valid_mask
        if tau < 0.50:
            if res_q[quiet_mask].size > 0:
                quiet_offset = np.quantile(res_q[quiet_mask], 1 - tau)
            else:
                quiet_offset = np.quantile(res_q, 1 - tau)
            if res_q[vol_mask].size > 0:
                vol_offset = np.quantile(res_q[vol_mask], 1 - tau)
            else:
                vol_offset = quiet_offset
            count_quiet = quiet_mask.sum()
            count_vol = vol_mask.sum()
            offsets[qi] = (count_quiet * quiet_offset + count_vol * vol_offset)
        else:
            if res_q[quiet_mask].size > 0:
                quiet_offset = np.quantile(res_q[quiet_mask], tau)
            else:
                quiet_offset = np.quantile(res_q, tau)
            if res_q[vol_mask].size > 0:
                vol_offset = np.quantile(res_q[vol_mask], tau)
            else:
                vol_offset = quiet_offset
            count_quiet = quiet_mask.sum()
            count_vol = vol_mask.sum()
            offsets[qi] = (count_quiet * quiet_offset + count_vol * vol_offset)
    else:
        if tau < 0.50:
            offsets[qi] = np.quantile(res_q, 1 - tau)
        elif tau > 0.50:
            offsets[qi] = np.quantile(res_q, tau)
        else:
            offsets[qi] = 0.0

adjusted_test = preds_test + offsets
adjusted_test[:, quantiles.index(0.50)] += median_bias
adjusted_test = isotonic_non_crossing(adjusted_test, quantiles)

losses = [mean_pinball_loss(y_test, adjusted_test[:, qi], alpha=tau) for qi, tau
```

```
            total_loss.append(np.mean(losses))

            start += step
            fold_count += 1

    return float(np.mean(total_loss))


study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=30)

print('Best parameters:', study.best_params)
print('Best average pinball loss:', study.best_value)

best_params = study.best_params
import json
with open('qrf_tuned_params.json', 'w') as f:
    json.dump(best_params, f)

best_params



import json
try:
    with open('qrf_tuned_params.json') as f:
        best_params = json.load(f)
except FileNotFoundError:
    best_params = {'n_estimators': 1000, 'max_features': 'sqrt', 'min_samples_leaf': 10, 'ma

quantiles = [0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95]
pred_records = []
pinball_records = []

for token in df['token'].unique():
    df_tok = df[df['token'] == token].reset_index(drop=True)
    n = len(df_tok)
    start = 0
    fold_idx = 0
    while start + train_len + cal_len + test_len <= n:
        train_slice = slice(start, start + train_len)
        cal_slice = slice(start + train_len, start + train_len + cal_len)
        test_slice = slice(start + train_len + cal_len, start + train_len + cal_len + test_l

        df_train = df_tok.iloc[train_slice]
        df_cal = df_tok.iloc[cal_slice]
        df_test = df_tok.iloc[test_slice]

        X_train = df_train[feature_cols]
        y_train = df_train[target_col]
        X_cal = df_cal[feature_cols]
```

```
y_cal = df_cal[target_col]
X_test = df_test[feature_cols]
y_test = df_test[target_col]

weights = compute_decay_weights(len(y_train), half_life=60)

model = RandomForestQuantileRegressor(
    n_estimators=1052,
    min_samples_leaf=6,
    max_features=0.9772503234610418,
    max_depth=26,
    bootstrap=True,
    random_state=42,
    n_jobs=-1
)

pipe = Pipeline([
    ('preprocess', preprocessor),
    ('qrf', model)
])

pipe.fit(X_train, y_train, qrf__sample_weight=weights)


preds_cal = np.array(pipe.predict(X_cal, quantiles=quantiles))
preds_test = np.array(pipe.predict(X_test, quantiles=quantiles))

residuals = y_cal.values.reshape(-1, 1) - preds_cal

if len(imputation_mask_cols) > 0:
    imputed_counts = df_cal[imputation_mask_cols].sum(axis=1)
    valid_mask = imputed_counts / len(imputation_mask_cols) < 0.3
else:
    valid_mask = np.ones(len(df_cal), dtype=bool)

#compute regime-aware   on calibration residuals
regime_labels = resolve_regime_labels(df_cal)  # <- uses your numeric vol_regime sa
offsets = np.zeros(len(quantiles))
median_bias = np.median(residuals[valid_mask, quantiles.index(0.50)])

for qi, tau in enumerate(quantiles):
    res_all = winsorize_residuals(residuals[valid_mask, qi])

    if tau in [0.05, 0.10, 0.90, 0.95]:
        quiet_mask = (regime_labels == "quiet") & valid_mask
        vol_mask   = (regime_labels == "volatile") & valid_mask

        def qtau(arr, t=tau, fallback=res_all):
            return np.quantile(winsorize_residuals(arr), t) if arr.size > 0 else np

        quiet_off = qtau(residuals[quiet_mask, qi])
```

```
                vol_off   = qtau(residuals[vol_mask,   qi])
                wq, wv = quiet_mask.sum(), vol_mask.sum()

        #if one side is missing, this gracefully reduces to the other / global
                denom = wq + wv
                if denom == 0:
                    offsets[qi] = np.quantile(res_all, tau)
                else:
                    offsets[qi] = (wq * quiet_off + wv * vol_off) / denom
            else:
                offsets[qi] = np.quantile(res_all, tau)

        adjusted_test = preds_test + offsets
        adjusted_test[:, quantiles.index(0.50)] += median_bias
        adjusted_test = isotonic_non_crossing(adjusted_test, quantiles)


        for qi, tau in enumerate(quantiles):
            loss = mean_pinball_loss(y_test, adjusted_test[:, qi], alpha=tau)
            pinball_records.append({
                'token': token,
                'fold': fold_idx,
                'tau': tau,
                'pinball_loss': loss
            })

        for i, row in df_test.iterrows():
            rec = {
                'token': token,
                'timestamp': row['timestamp'],
                'fold': fold_idx,
                'y_true': row[target_col]
            }
            for qi, tau in enumerate(quantiles):
                rec[f'q{int(tau*100)}'] = adjusted_test[i - test_slice.start, qi]
            pred_records.append(rec)

        start += step
        fold_idx += 1

pred_df = pd.DataFrame(pred_records)
pinball_df = pd.DataFrame(pinball_records)
avg_pinball = pinball_df.groupby('tau')['pinball_loss'].mean().reset_index().rename(columns=

pred_df.to_csv('qrf_v2_tuned_preds.csv', index=False)
pinball_df.to_csv('qrf_v2_tuned_pinball.csv', index=False)
avg_pinball.to_csv('qrf_v3_tuned_avg_pinball.csv', index=False)

avg_pinball
```

Feature Importance analysis using v3

```
##Feature importance using the final tuned model (with decay weights)

#Prepare full dataset
X_full = df[feature_cols]
y_full = df[target_col]
weights_full = compute_decay_weights(len(y_full), half_life=60)

#Build the tuned model
final_model = RandomForestQuantileRegressor(
    n_estimators=int(best_params.get('n_estimators', 1000)),
    min_samples_leaf=int(best_params.get('min_samples_leaf', 10)),
    max_features=best_params.get('max_features', 'sqrt'),
    max_depth=best_params.get('max_depth', None),
    bootstrap=True,
    random_state=42,
    n_jobs=-1
)

final_pipe = Pipeline([
    ('preprocess', preprocessor),
    ('qrf', final_model)
])

#Fit the model, attempting to include sample weights.
#If the underlying estimator does not support sample_weight, fit without it.
try:
    final_pipe.fit(X_train, y_train, qrf__sample_weight=weights)
except TypeError:
    final_pipe.fit(X_full, y_full)

#Access the fitted forest
forest = final_pipe.named_steps['qrf']

#Get one-hot and numeric feature names
cat_feature_names = final_pipe.named_steps['preprocess'].named_transformers_['cat'].get_feat
num_feature_names = numeric_cols
all_feature_names = list(cat_feature_names) + num_feature_names

#Retrieve MDI importances from the forest
importances = forest.feature_importances_

#Aggregate importances back to original feature names
from collections import defaultdict
agg_importance = defaultdict(float)
for name, imp in zip(all_feature_names, importances):
    original = name.split('_')[0] if name in cat_feature_names else name
    agg_importance[original] += imp

importances_df = (
    pd.DataFrame({'feature': agg_importance.keys(), 'importance': agg_importance.values()})
      .sort_values('importance', ascending=False)
```

```
)

#Save and display
importances_df.to_csv('qrf_v2_tuned_feature_importances.csv', index=False)
importances_df
```

QRF Final Model

```
##Feature importance using the final tuned model (with decay weights)

#Prepare full dataset
X_full = df[feature_cols]
y_full = df[target_col]
weights_full = compute_decay_weights(len(y_full), half_life=60)

#Build the tuned model
final_model = RandomForestQuantileRegressor(
    n_estimators=int(best_params.get('n_estimators', 1000)),
    min_samples_leaf=int(best_params.get('min_samples_leaf', 10)),
    max_features=best_params.get('max_features', 'sqrt'),
    max_depth=best_params.get('max_depth', None),
    bootstrap=True,
    random_state=42,
    n_jobs=-1
)

final_pipe = Pipeline([
    ('preprocess', preprocessor),
    ('qrf', final_model)
])

#Fit the model, attempting to include sample weights.
#If the underlying estimator does not support sample_weight, fit without it.
try:
    final_pipe.fit(X_train, y_train, qrf__sample_weight=weights)
except TypeError:
    final_pipe.fit(X_full, y_full)

#Access the fitted forest
forest = final_pipe.named_steps['qrf']

#Get one-hot and numeric feature names
cat_feature_names = final_pipe.named_steps['preprocess'].named_transformers_['cat'].get_feat
num_feature_names = numeric_cols
all_feature_names = list(cat_feature_names) + num_feature_names

#Retrieve MDI importances from the forest
importances = forest.feature_importances_

#Aggregate importances back to original feature names
from collections import defaultdict
```

```
agg_importance = defaultdict(float)
for name, imp in zip(all_feature_names, importances):
    original = name.split('_')[0] if name in cat_feature_names else name
    agg_importance[original] += imp

importances_df = (
    pd.DataFrame({'feature': agg_importance.keys(), 'importance': agg_importance.values()})
      .sort_values('importance', ascending=False)
)

#Save and display
importances_df.to_csv('qrf_v2_tuned_feature_importances.csv', index=False)
importances_df


import optuna

def objective(trial: optuna.Trial) -> float:
    n_estimators = trial.suggest_int('n_estimators', 600, 2000)
    max_features_choice = trial.suggest_categorical('max_features_choice', ['sqrt', 'log2',
    max_features = trial.suggest_float('max_features', 0.3, 1.0) if max_features_choice ==
    min_samples_leaf = trial.suggest_int('min_samples_leaf', 5, 60)
    max_depth = trial.suggest_categorical('max_depth', [None] + list(range(6, 29, 2)))

    total_loss = []
    params = dict(n_estimators=n_estimators, max_features=max_features,
                  min_samples_leaf=min_samples_leaf, max_depth=max_depth)

    for token in tokens_to_use:
        df_tok = df[df['token'] == token].reset_index(drop=True)
        n, start, fold_count = len(df_tok), 0, 0

        while start + train_len + cal_len + test_len <= n and fold_count < 10:
            tr = slice(start, start + train_len)
            ca = slice(start + train_len, start + train_len + cal_len)
            te = slice(start + train_len + cal_len, start + train_len + cal_len + test_len)

            df_train, df_cal, df_test = df_tok.iloc[tr], df_tok.iloc[ca], df_tok.iloc[te]
            X_train, y_train = df_train[feature_cols], df_train[target_col]
            X_cal,   y_cal   = df_cal[feature_cols],   df_cal[target_col]
            X_test,  y_test  = df_test[feature_cols],  df_test[target_col]

            pipe = Pipeline([('preprocess', preprocessor),
                             ('qrf', RandomForestQuantileRegressor(
                                  n_estimators=params["n_estimators"],
                                  min_samples_leaf=params["min_samples_leaf"],
                                  max_features=params["max_features"],
                                  max_depth=params["max_depth"],
                                  bootstrap=True, random_state=42, n_jobs=-1))])

            pipe.fit(X_train, y_train, qrf__sample_weight=compute_decay_weights(len(y_train)
```

```
            preds_cal  = np.array(pipe.predict(X_cal,  quantiles=quantiles))
            preds_test = np.array(pipe.predict(X_test, quantiles=quantiles))
            residuals  = y_cal.values.reshape(-1, 1) - preds_cal

            cal_mask = make_cal_mask(df_cal, y_cal, imputation_mask_cols)

            # offsets
            regime_labels = resolve_regime_labels(df_cal)
            offsets = np.zeros(len(quantiles), dtype=float)
            for qi, tau in enumerate(quantiles):
                res_all = winsorize_residuals_nan(residuals[cal_mask, qi])
                if tau in (0.05, 0.10, 0.90, 0.95):
                    quiet_mask = ((regime_labels == "quiet").to_numpy()) & cal_mask
                    vol_mask   = ((regime_labels == "volatile").to_numpy()) & cal_mask
                    quiet_res = winsorize_residuals_nan(residuals[quiet_mask, qi])
                    vol_res   = winsorize_residuals_nan(residuals[vol_mask,   qi])
                    wq, wv = quiet_res.size, vol_res.size
                    if (wq + wv) == 0:
                        offsets[qi] = nanquant(res_all, tau, fallback=0.0)
                    else:
                        q_off = nanquant(quiet_res, tau, fallback=nanquant(res_all, tau))
                        v_off = nanquant(vol_res,   tau, fallback=nanquant(res_all, tau))
                        offsets[qi] = (wq * q_off + wv * v_off) / (wq + wv)
                else:
                    offsets[qi] = nanquant(res_all, tau, fallback=0.0)

            adj_cal  = isotonic_non_crossing(preds_cal  + offsets, quantiles)
            adj_test = isotonic_non_crossing(preds_test + offsets, quantiles)

            delta80 = split_conformal_delta_two_sided(y_cal.values, adj_cal[:, i10], adj_cal
            delta90 = split_conformal_delta_two_sided(y_cal.values, adj_cal[:, i05], adj_cal

            adj_test[:, i10] -= delta80; adj_test[:, i90] += delta80
            adj_test[:, i05] -= delta90; adj_test[:, i95] += delta90

            adj_test = isotonic_non_crossing(adj_test, quantiles)

            losses = [mean_pinball_loss(y_test, adj_test[:, qi], alpha=tau) for qi, tau in e
            total_loss.append(float(np.mean(losses)))

            start += step; fold_count += 1

    return float(np.mean(total_loss))

#Suggested sampler/pruner to speed up
sampler = optuna.samplers.TPESampler(seed=42, n_startup_trials=8)
pruner  = optuna.pruners.MedianPruner(n_warmup_steps=6)
study = optuna.create_study(direction='minimize', sampler=sampler, pruner=pruner)
study.optimize(objective, n_trials=40)
```

```
print('Best parameters:', study.best_params)
print('Best average pinball loss:', study.best_value)

with open('qrf_tuned_params.json','w') as f:
    json.dump(study.best_params, f)


#=== Load tuned params or fallback ==========================================
try:
    with open('qrf_tuned_params.json') as f:
        best_params = json.load(f)
except FileNotFoundError:
    best_params = {'n_estimators': 1000, 'max_features': 'sqrt', 'min_samples_leaf': 10, 'ma

def build_qrf(params):
    return RandomForestQuantileRegressor(
        n_estimators=params.get("n_estimators", 1000),
        min_samples_leaf=params.get("min_samples_leaf", 10),
        max_features=params.get("max_features", "sqrt"),
        max_depth=params.get("max_depth", None),
        bootstrap=True,
        random_state=42,
        n_jobs=-1
    )

pred_records, pinball_records = [], []

for token in tokens_to_use:
    df_tok = df[df['token'] == token].reset_index(drop=True)
    n, start, fold_idx = len(df_tok), 0, 0

    while start + train_len + cal_len + test_len <= n:
        tr = slice(start, start + train_len)
        ca = slice(start + train_len, start + train_len + cal_len)
        te = slice(start + train_len + cal_len, start + train_len + cal_len + test_len)

        df_train, df_cal, df_test = df_tok.iloc[tr], df_tok.iloc[ca], df_tok.iloc[te]
        X_train, y_train = df_train[feature_cols], df_train[target_col]
        X_cal,   y_cal   = df_cal[feature_cols],   df_cal[target_col]
        X_test,  y_test  = df_test[feature_cols],  df_test[target_col]

        pipe = Pipeline([
            ('preprocess', preprocessor),
            ('qrf', build_qrf(best_params))
        ])

        pipe.fit(X_train, y_train, qrf__sample_weight=compute_decay_weights(len(y_train), 60

        preds_cal  = np.array(pipe.predict(X_cal,  quantiles=quantiles))
        preds_test = np.array(pipe.predict(X_test, quantiles=quantiles))
        residuals  = y_cal.values.reshape(-1, 1) - preds_cal
```

```
#---------- valid calibration rows ----------
cal_mask = make_cal_mask(df_cal, y_cal, imputation_mask_cols)

#---------- regime-aware residual quantile offsets (correct  =Q ) ----------
regime_labels = resolve_regime_labels(df_cal)  # "quiet"|"mid"|"volatile"
offsets = np.zeros(len(quantiles), dtype=float)

for qi, tau in enumerate(quantiles):
    res_all = winsorize_residuals_nan(residuals[cal_mask, qi])
    if tau in (0.05, 0.10, 0.90, 0.95):
        quiet_mask = ((regime_labels == "quiet").to_numpy()) & cal_mask
        vol_mask   = ((regime_labels == "volatile").to_numpy()) & cal_mask

        quiet_res = winsorize_residuals_nan(residuals[quiet_mask, qi])
        vol_res   = winsorize_residuals_nan(residuals[vol_mask,   qi])

        wq, wv = quiet_res.size, vol_res.size
        if (wq + wv) == 0:
            offsets[qi] = nanquant(res_all, tau, fallback=0.0)
        else:
            q_off = nanquant(quiet_res, tau, fallback=nanquant(res_all, tau))
            v_off = nanquant(vol_res,   tau, fallback=nanquant(res_all, tau))
            offsets[qi] = (wq * q_off + wv * v_off) / (wq + wv)
    else:
        offsets[qi] = nanquant(res_all, tau, fallback=0.0)

#apply offsets to CAL/TEST, then enforce monotonicity
adj_cal  = isotonic_non_crossing(preds_cal  + offsets, quantiles)
adj_test = isotonic_non_crossing(preds_test + offsets, quantiles)

#---------- split-conformal widening for two-sided bands ----------
delta80 = split_conformal_delta_two_sided(y_cal.values, adj_cal[:, i10], adj_cal[:,
delta90 = split_conformal_delta_two_sided(y_cal.values, adj_cal[:, i05], adj_cal[:,

adj_test[:, i10] -= delta80;  adj_test[:, i90] += delta80
adj_test[:, i05] -= delta90;  adj_test[:, i95] += delta90

#final monotonic guard
adj_test = isotonic_non_crossing(adj_test, quantiles)

#---------- record metrics & predictions ----------
for qi, tau in enumerate(quantiles):
    loss = mean_pinball_loss(y_test, adj_test[:, qi], alpha=tau)
    pinball_records.append({"token": token, "fold": fold_idx, "tau": tau, "pinball_l

for i, row in df_test.iterrows():
    rec = {"token": token, "timestamp": row["timestamp"], "fold": fold_idx, "y_true"
    for qi, tau in enumerate(quantiles):
        rec[f"q{int(tau*100)}"] = adj_test[i - te.start, qi]
    pred_records.append(rec)
```

```
        start   += step
        fold_idx += 1

#=== Save outputs
pred_df    = pd.DataFrame(pred_records)
pinball_df = pd.DataFrame(pinball_records)
avg_pinball = (pinball_df.groupby('tau')['pinball_loss']
               .mean().reset_index()
               .rename(columns={'pinball_loss':'avg_pinball_loss'}))

pred_df.to_csv('qrf_v2_tuned_preds.csv', index=False)
pinball_df.to_csv('qrf_v2_tuned_pinball.csv', index=False)
avg_pinball.to_csv('qrf_v2_tuned_avg_pinball.csv', index=False)
print("Saved: qrf_v2_tuned_preds.csv, qrf_v2_tuned_pinball.csv, qrf_v4_tuned_avg_pinball.csv
```

QRF Robustness Testing Model

```
#=== Robustness harness prerequisites =
import json, math, numpy as np, pandas as pd
from sklearn.pipeline import Pipeline
from quantile_forest import RandomForestQuantileRegressor
from sklearn.isotonic import IsotonicRegression
from sklearn.metrics import mean_pinball_loss
from scipy.stats import iqr

#data prerequisites (asserts so we fail fast if missing)
assert 'df' in globals(), "df not found"
assert 'feature_cols' in globals() and 'target_col' in globals(), "feature_cols/target_col n

#If you didn't define a preprocessor earlier, use passthrough.
try:
    preprocessor
except NameError:
    preprocessor = 'passthrough'

#If imputation mask list not defined, use empty list
imputation_mask_cols = imputation_mask_cols if 'imputation_mask_cols' in globals() else []

#--- Tuned params (load if available)
try:
    with open('qrf_tuned_params.json') as f:
        best_params = json.load(f)
except FileNotFoundError:
    best_params = {'n_estimators': 1000, 'min_samples_leaf': 10,
                   'max_features': 'sqrt', 'max_depth': None}
for k, v in {'n_estimators':1000, 'min_samples_leaf':10,
             'max_features':'sqrt', 'max_depth':None}.items():
    best_params.setdefault(k, v)
print("Using QRF params:", best_params)
```

```
#--- rolling config
train_len, cal_len, test_len, step = 120, 24, 6, 6
TAUS = [0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95]
QI = {t:i for i, t in enumerate(TAUS)}
i05, i10, i25, i50, i75, i90, i95 = [QI[t] for t in TAUS]

#optional caps (set None to disable)
MAX_FOLDS_PER_TOKEN = None        # e.g., 10 to speed up
TOKENS_SUBSET = None              # e.g., ["SOL", "BONK"] to subset

#--- helpers (defined only if missing)
def _define_if_missing(name, fn):
    if name not in globals(): globals()[name] = fn

_define_if_missing('compute_decay_weights',
    lambda n, half_life=60.0: (lambda idx, k: (np.exp(-idx/k) / np.exp(-idx/k).sum()))(
        np.arange(n)[::-1], half_life / math.log(2.0)
    )
)

def _make_cal_mask(df_cal, y_cal, imputation_mask_cols, thresh=0.30):
    mask = np.isfinite(y_cal.values)
    if imputation_mask_cols:
        imp = df_cal[imputation_mask_cols].sum(axis=1).to_numpy()
        mask &= (imp / len(imputation_mask_cols)) < thresh
    return mask
_define_if_missing('make_cal_mask', _make_cal_mask)

def _winsorize_residuals_nan(arr):
    arr = np.asarray(arr, dtype=float)
    arr = arr[np.isfinite(arr)]
    if arr.size == 0: return arr
    width = iqr(arr) if np.isfinite(iqr(arr)) else np.nanstd(arr)
    width = width if (width and width > 0) else np.nanstd(arr)
    med = np.nanmedian(arr)
    lo, hi = med - 5*width, med + 5*width
    return np.clip(arr, lo, hi)
_define_if_missing('winsorize_residuals_nan', _winsorize_residuals_nan)

_define_if_missing('nanquant',
    lambda arr, q, fallback=0.0: float(np.nanquantile(np.asarray(arr, float)[np.isfinite(arr
                                 if np.isfinite(np.asarray(arr, float)).any() else float(fal
)

def _isotonic_non_crossing(preds, taus):
    taus_arr = np.asarray(taus, float)
    out = np.empty_like(preds, float)
    col_med = np.nanmedian(preds, axis=0)
    col_med = np.where(np.isfinite(col_med), col_med, 0.0)
    ir = IsotonicRegression(increasing=True, out_of_bounds='clip')
    for r in range(preds.shape[0]):
```

```
        row = preds[r, :].astype(float)
        finite = np.isfinite(row)
        if not finite.any():
            row_filled = col_med.copy()
        elif finite.sum() < row.size:
            row_filled = np.interp(taus_arr, taus_arr[finite], row[finite])
        else:
            row_filled = row
        row_filled = np.where(np.isfinite(row_filled), row_filled, col_med)
        out[r, :] = ir.fit_transform(taus_arr, row_filled)
    return out
_define_if_missing('isotonic_non_crossing', _isotonic_non_crossing)

def _split_conformal_delta_two_sided(y, q_lo, q_hi, coverage):
    y, q_lo, q_hi = map(lambda a: np.asarray(a, float), (y, q_lo, q_hi))
    mask = np.isfinite(y) & np.isfinite(q_lo) & np.isfinite(q_hi)
    if mask.sum() == 0: return 0.0
    s = np.maximum(q_lo[mask] - y[mask], y[mask] - q_hi[mask])
    n = s.size
    k = int(np.ceil((n + 1) * coverage)) - 1
    k = max(0, min(k, n - 1))
    return float(np.partition(s, k)[k])
_define_if_missing('split_conformal_delta_two_sided', _split_conformal_delta_two_sided)

def _resolve_regime_labels(df_fold: pd.DataFrame) -> pd.Series:
    if "vol_regime" in df_fold.columns:
        reg = df_fold["vol_regime"]
        if pd.api.types.is_numeric_dtype(reg):
            out = pd.Series(np.where(reg >= 3, "volatile",
                                     np.where(reg <= 1, "quiet", "mid")),
                            index=reg.index, dtype="object")
            out[reg.isna()] = "mid"
            return out
        m = {"low":"quiet","quiet":"quiet","calm":"quiet","mid":"mid","normal":"mid",
             "high":"volatile","volatile":"volatile","wild":"volatile"}
        return reg.astype(str).str.lower().map(m).fillna("mid")
    proxy = next((c for c in ["gk_vol_36h","parkinson_vol_36h","vol_std_7bar","downside_vol
                  if c in df_fold.columns), None)
    if proxy:
        v = df_fold[proxy]; q1, q2 = v.quantile([0.33, 0.66])
        out = pd.Series(np.where(v >= q2, "volatile", np.where(v <= q1, "quiet", "mid")),
                        index=v.index, dtype="object")
        out[v.isna()] = "mid"; return out
    return pd.Series(["mid"] * len(df_fold), index=df_fold.index, dtype="object")
_define_if_missing('resolve_regime_labels', _resolve_regime_labels)

#--- pipeline builder
def build_pipe(params: dict):
    return Pipeline([
        ('preprocess', preprocessor),
        ('qrf', RandomForestQuantileRegressor(
```

```
            n_estimators=params.get('n_estimators', 1000),
            min_samples_leaf=params.get('min_samples_leaf', 10),
            max_features=params.get('max_features', 'sqrt'),
            max_depth=params.get('max_depth', None),
            bootstrap=True, random_state=42, n_jobs=-1
        ))
    ])

#--- offsets computation (global vs regime-aware)
def compute_offsets(residuals, cal_mask, taus, regime_labels=None, mode='global'):
    offs = np.zeros(len(taus), float)
    for qi, tau in enumerate(taus):
        res_all = winsorize_residuals_nan(residuals[cal_mask, qi])
        if mode == 'regime' and regime_labels is not None and tau in (0.05, 0.10, 0.90, 0.95
            quiet_mask = ((regime_labels == "quiet").to_numpy()) & cal_mask
            vol_mask   = ((regime_labels == "volatile").to_numpy()) & cal_mask
            qres = winsorize_residuals_nan(residuals[quiet_mask, qi])
            vres = winsorize_residuals_nan(residuals[vol_mask,   qi])
            wq, wv = qres.size, vres.size
            if wq + wv == 0:
                offs[qi] = nanquant(res_all, tau, 0.0)
            else:
                q_off = nanquant(qres, tau, nanquant(res_all, tau))
                v_off = nanquant(vres, tau, nanquant(res_all, tau))
                offs[qi] = (wq * q_off + wv * v_off) / (wq + wv)
        else:
            offs[qi] = nanquant(res_all, tau, 0.0)
    return offs

#--- pooled stats for 'calibration_scope="pooled"'
def collect_pooled_stats(df, params, *, use_decay=True, half_life=60,
                         offsets_mode='regime', isotonic=True):
    all_res, all_y_cal, all_q10, all_q90, all_q05, all_q95 = [], [], [], [], [], []
    for tok, df_tok in df.groupby('token'):
        df_tok = df_tok.reset_index(drop=True)
        n, start = len(df_tok), 0
        while start + train_len + cal_len + test_len <= n:
            tr = slice(start, start + train_len)
            ca = slice(start + train_len, start + train_len + cal_len)
            te = slice(start + train_len + cal_len, start + train_len + cal_len + test_len)

            df_tr, df_ca = df_tok.iloc[tr], df_tok.iloc[ca]
            X_tr, y_tr = df_tr[feature_cols], df_tr[target_col]
            X_ca, y_ca = df_ca[feature_cols], df_ca[target_col]

            pipe = build_pipe(params)
            sw = compute_decay_weights(len(y_tr), half_life) if use_decay else None
            fit_kwargs = {'qrf__sample_weight': sw} if sw is not None else {}
            pipe.fit(X_tr, y_tr, **fit_kwargs)

            cal_hat = np.array(pipe.predict(X_ca, quantiles=TAUS))
```

```
            if isotonic:
                cal_hat = isotonic_non_crossing(cal_hat, TAUS)

            residuals = y_ca.values.reshape(-1,1) - cal_hat
            cal_mask  = make_cal_mask(df_ca, y_ca, imputation_mask_cols)
            regs      = resolve_regime_labels(df_ca) if offsets_mode == 'regime' else None

            all_res.append((residuals, cal_mask, regs))
            all_y_cal.append(y_ca.values)
            all_q10.append(cal_hat[:, i10]); all_q90.append(cal_hat[:, i90])
            all_q05.append(cal_hat[:, i05]); all_q95.append(cal_hat[:, i95])

            start += step

    #offsets: pool residuals across folds
    if offsets_mode == 'regime':
        #weight by counts across folds
        R = np.zeros(len(TAUS))
        W = np.zeros(len(TAUS))
        #compute regime-aware offset per fold then average weighted by #valid rows
        offs_list, wts = [], []
        for residuals, cal_mask, regs in all_res:
            offs = compute_offsets(residuals, cal_mask, TAUS, regime_labels=regs, mode='regi
            offs_list.append(offs); wts.append(int(cal_mask.sum()))
        offs = np.average(np.vstack(offs_list), axis=0, weights=wts) if offs_list else np.ze
    else:
        #global offsets from concatenated residuals
        if not all_res:
            offs = np.zeros(len(TAUS))
        else:
            res_stack = []
            for residuals, cal_mask, _ in all_res:
                res_stack.append(residuals[cal_mask, :])
            res_stack = np.vstack(res_stack)
            offs = np.array([nanquant(_winsorize_residuals_nan(res_stack[:, qi]), tau, 0.0)
                             for qi, tau in enumerate(TAUS)])

    #conformal deltas (pooled)
    y_all  = np.concatenate(all_y_cal) if all_y_cal else np.array([])
    q10_all= np.concatenate(all_q10) if all_q10 else np.array([])
    q90_all= np.concatenate(all_q90) if all_q90 else np.array([])
    q05_all= np.concatenate(all_q05) if all_q05 else np.array([])
    q95_all= np.concatenate(all_q95) if all_q95 else np.array([])

    d80 = split_conformal_delta_two_sided(y_all, q10_all, q90_all, 0.80) if y_all.size else
    d90 = split_conformal_delta_two_sided(y_all, q05_all, q95_all, 0.90) if y_all.size else
    return {'offsets': offs, 'd80': d80, 'd90': d90}

rows = [run_variant(df, best_params, **cfg) for cfg in variants]
robust = pd.DataFrame(rows)
robust.to_csv('tbl_robustness_summary.csv', index=False)
```

```
robust
```

E.3 Model Analysis, Testing and Application to Trading

See the full notebooks here: - Comparison of Models, using Pinball loss, DM test and more. - Validation Tests on QRF, addition tests/analysis, Risk aware sizing backtest and Quantile Prediction Figures

- **Computing pinball loss by tau, and coverage of all models**

```
import pandas as pd
import numpy as np
from sklearn.metrics import mean_pinball_loss

#CONFIG
qrf_path = 'qrf_v2_tuned_preds.csv'
lqr_path = 'lqr_pred_paths_full.csv'
lgb_path = 'lgb_extended_preds.csv'

#Standard quantile set and canonical column names
TAUS = [0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95]
QCOLS = {0.05:'q05_pred', 0.10:'q10_pred', 0.25:'q25_pred',
         0.50:'q50_pred', 0.75:'q75_pred', 0.90:'q90_pred', 0.95:'q95_pred'}

#HELPERS (robust)
def find_and_rename_id_cols(df):
    """Ensure columns: token, timestamp, y_true (if present)."""
    #token
    tok_candidates = [c for c in df.columns if c.lower() in ('token','symbol','asset')]
    if tok_candidates and 'token' not in df.columns:
        df = df.rename(columns={tok_candidates[0]: 'token'})
    #timestamp
    ts_candidates = [c for c in df.columns if c.lower() in ('timestamp','time','date','datet
    if ts_candidates and 'timestamp' not in df.columns:
        df = df.rename(columns={ts_candidates[0]: 'timestamp'})
    #y_true (optional)
    if 'y_true' not in df.columns:
        y_candidates = [c for c in df.columns if c.lower() in ('y_true','y','target','ret_72
        if y_candidates:
            df = df.rename(columns={y_candidates[0]: 'y_true'})
    return df

def normalize_quantile_cols(df):
    """
    Map any common variants to canonical qXX_pred names.
    Handles: q5, q05, q_05, q05_pred, q5_pred, q50, q50_pred, etc.
    Idempotent if already standard.
    """
```

```
    rename = {}
    for tau, canon in QCOLS.items():
        two = f"{int(tau*100):02d}"    # '05','10',...
        candidates = [
            f"q{two}", f"q{int(tau*100)}", f"q_{two}",
            f"q{two}_pred", f"q{int(tau*100)}_pred", f"q_{two}_pred"
        ]
        # Also accept bare 'q5' (tau=0.05)
        if tau == 0.05: candidates += ['q5','q5_pred']
        for c in candidates:
            if c in df.columns:
                rename[c] = canon
                break
    return df.rename(columns=rename)

def coerce_types(df):
    if 'timestamp' in df.columns:
        df['timestamp'] = pd.to_datetime(df['timestamp'], errors='coerce', utc=True)
    if 'token' in df.columns:
        df['token'] = df['token'].astype(str)
    if 'y_true' in df.columns:
        df['y_true'] = pd.to_numeric(df['y_true'], errors='coerce')
    #Quantile preds numeric
    for col in QCOLS.values():
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors='coerce')
    return df

def attach_y_true_if_missing(pred_df, y_source_df):
    """If pred_df lacks y_true, merge it from y_source_df on (token,timestamp)."""
    if 'y_true' in pred_df.columns:
        return pred_df
    cols = ['token','timestamp','y_true']
    if not set(cols[:2]).issubset(y_source_df.columns):
        raise ValueError("y_source_df must have token and timestamp to attach y_true.")
    merged = pred_df.merge(y_source_df[cols], on=['token','timestamp'], how='left', validate
    return merged

def compute_coverage(df):
    """Return dict with 80% and 90% coverage, using only rows where bounds & y_true are fini
    out = {}
    #80%: q10 - q90
    if {'y_true','q10_pred','q90_pred'}.issubset(df.columns):
        m = np.isfinite(df['y_true']) & np.isfinite(df['q10_pred']) & np.isfinite(df['q90_pr
        if m.any():
            out['cov80'] = np.mean((df.loc[m,'y_true'] >= df.loc[m,'q10_pred']) &
                                   (df.loc[m,'y_true'] <= df.loc[m,'q90_pred']))
        else:
            out['cov80'] = np.nan
    else:
        out['cov80'] = np.nan
```

```
    #90%: q05 - q95
    if {'y_true','q05_pred','q95_pred'}.issubset(df.columns):
        m = np.isfinite(df['y_true']) & np.isfinite(df['q05_pred']) & np.isfinite(df['q95_pr
        if m.any():
            out['cov90'] = np.mean((df.loc[m,'y_true'] >= df.loc[m,'q05_pred']) &
                                   (df.loc[m,'y_true'] <= df.loc[m,'q95_pred']))
        else:
            out['cov90'] = np.nan
    else:
        out['cov90'] = np.nan
    return out

def compute_pinball_by_tau(df):
    """Return DataFrame with pinball loss per  using sklearn's mean_pinball_loss; NaN-safe
    rows = []
    for tau, col in QCOLS.items():
        if {'y_true', col}.issubset(df.columns):
            m = np.isfinite(df['y_true']) & np.isfinite(df[col])
            if m.any():
                loss = mean_pinball_loss(df.loc[m,'y_true'].to_numpy(float),
                                         df.loc[m,col].to_numpy(float),
                                         alpha=tau)
                rows.append({'tau': tau, 'pinball_loss': float(loss)})
            else:
                rows.append({'tau': tau, 'pinball_loss': np.nan})
        else:
            rows.append({'tau': tau, 'pinball_loss': np.nan})
    return pd.DataFrame(rows)

#LOAD & CLEAN
qrf_df = pd.read_csv(qrf_path)
lqr_df = pd.read_csv(lqr_path)
lgb_df = pd.read_csv(lgb_path)

#Standardize ids & quantile columns
qrf_df = coerce_types(normalize_quantile_cols(find_and_rename_id_cols(qrf_df)))
lqr_df = coerce_types(normalize_quantile_cols(find_and_rename_id_cols(lqr_df)))
lgb_df = coerce_types(normalize_quantile_cols(find_and_rename_id_cols(lgb_df)))

#Attach y_true to LQR/LGB if missing using QRF as source-of-truth for (token,timestamp)->y_t
lqr_df = attach_y_true_if_missing(lqr_df, qrf_df)
lgb_df = attach_y_true_if_missing(lgb_df, qrf_df)

#Keep only necessary columns for each model
cols_needed = ['token','timestamp','y_true'] + list(QCOLS.values())
qrf_use = qrf_df[[c for c in cols_needed if c in qrf_df.columns]].copy()
lqr_use = lqr_df[[c for c in cols_needed if c in lqr_df.columns]].copy()
lgb_use = lgb_df[[c for c in cols_needed if c in lgb_df.columns]].copy()

#METRICS PER MODEL
coverage_records = []
```

```
pinball_records = []

for model, dfm in [('QRF', qrf_use), ('LQR', lqr_use), ('LightGBM', lgb_use)]:
    cov = compute_coverage(dfm)
    coverage_records.append({'model': model, 'interval': '80%', 'coverage': cov['cov80']})
    coverage_records.append({'model': model, 'interval': '90%', 'coverage': cov['cov90']})

    pb = compute_pinball_by_tau(dfm)
    pb['model'] = model
    pinball_records.append(pb)

coverage_df = pd.DataFrame(coverage_records)
pinball_df = pd.concat(pinball_records, ignore_index=True)

#Sanity: pinball must be >= 0 (allowing tiny numerical eps)

#Optional: pretty print / save
print(coverage_df)
print(pinball_df.sort_values(['model','tau']))

coverage_df.to_csv('tbl_coverage_summary.csv', index=False)
pinball_df.to_csv('tbl_pinball_by_tau.csv', index=False)
```

- **DM Tests comparing QRF to LQR and LightGBM for each quantile**

```
#=========================
#Diebold-Mariano table (QRF vs LQR/LGBM)
#=========================
import numpy as np
import pandas as pd

def _pstars(p):
    if p < 0.001: return "***"
    if p < 0.01:  return "**"
    if p < 0.05:  return "*"
    if p < 0.10:  return "†"
    return ""

def make_dm_table(DM_df: pd.DataFrame):
    """
    Formats DM results into an academic-quality table.
    Assumes DM statistic is built from (loss_QRF - loss_other):
      • Negative DM  QRF lower expected pinball loss (better)
      • Positive DM  Comparator better
    """
    df = DM_df.copy()
    #Pretty  label
    df[" "] = df["tau"].apply(lambda t: f"{int(round(t*100)):02d}%")

    #Direction flags
    df["favours_vs_LQR"] = np.where(df["dm_stat_qrf_vs_lqr"] < 0, "QRF", "LQR")
```

```
    df["favours_vs_LGBM"] = np.where(df["dm_stat_qrf_vs_lgb"] < 0, "QRF", "LightGBM")

    #Select & rename columns for display
    out = (df[[" ",
               "dm_stat_qrf_vs_lqr","p_val_qrf_vs_lqr","favours_vs_LQR",
               "dm_stat_qrf_vs_lgb","p_val_qrf_vs_lgb","favours_vs_LGBM"]]
             .rename(columns={
                 "dm_stat_qrf_vs_lqr":"DM (QRF-LQR)",
                 "p_val_qrf_vs_lqr":"p (QRF-LQR)",
                 "dm_stat_qrf_vs_lgb":"DM (QRF-LGBM)",
                 "p_val_qrf_vs_lgb":"p (QRF-LGBM)",
                 "favours_vs_LQR":"Favours vs LQR",
                 "favours_vs_LGBM":"Favours vs LGBM"
             })
          )

    #Formatting
    fmt = {
        "DM (QRF-LQR)":  "{:+.3f}",
        "DM (QRF-LGBM)": "{:+.3f}",
        "p (QRF-LQR)":   lambda x: f"{x:.3f}{_pstars(x)}",
        "p (QRF-LGBM)":  lambda x: f"{x:.3f}{_pstars(x)}",
    }

    #Highlight significant cells; accent = significant & favours QRF
    def _highlight(data):
        styles = pd.DataFrame("", index=data.index, columns=data.columns)
        for dm_col, p_col, fav_col in [
            ("DM (QRF-LQR)","p (QRF-LQR)","Favours vs LQR"),
            ("DM (QRF-LGBM)","p (QRF-LGBM)","Favours vs LGBM"),
        ]:
            sig = data[p_col].astype(float) <= 0.05
            fav_qrf = sig & (data[fav_col] == "QRF")
            fav_other = sig & (data[fav_col] != "QRF")
            styles.loc[fav_qrf,  [dm_col,p_col,fav_col]]  = f"font-weight:700; color:{ACCENT
            styles.loc[fav_other,[dm_col,p_col,fav_col]]  = "font-weight:700; color:#B91C1C"
        return styles

    sty = (out.style
           .format(fmt)
           .apply(_highlight, axis=None)
           .set_properties(**{"text-align":"center"})
          )
    caption = ("Diebold-Mariano tests on pinball loss (per ). "
               "Negative DM  QRF lower expected loss; stars denote significance "
               "(*** <0.001, ** <0.01, * <0.05, † <0.10).")
    return _base_styler(sty, caption)

#--- Example usage in a Quarto cell ---
dm_tbl = make_dm_table(DM_df)
dm_tbl
```

- **HAC-robust Diebold–Mariano + per-token heatmap**

```
#================ DM utilities (run once) ================
import numpy as np, pandas as pd
from pathlib import Path
import matplotlib.pyplot as plt

RESULTS_DIR = Path("results"); RESULTS_DIR.mkdir(exist_ok=True)
TAUS = [0.05,0.10,0.25,0.50,0.75,0.90,0.95]
TAU2COL = {0.05:"q5",0.10:"q10",0.25:"q25",0.50:"q50",0.75:"q75",0.90:"q90",0.95:"q95"}

def pinball_loss_vec(y, q, tau):
    diff = y - q
    return np.maximum(tau*diff, (tau-1)*diff)

def newey_west_var(d, lag=5):
    """Bartlett kernel HAC variance of mean(d). Returns var(mean(d))."""
    d = np.asarray(d, dtype=float)
    d = d[np.isfinite(d)]
    n = d.size
    if n <= 1:
        return np.nan
    d = d - d.mean()
    gamma0 = np.dot(d, d) / n
    s = gamma0
    for k in range(1, min(lag, n-1)+1):
        w = 1 - k/(lag+1)
        gamma_k = np.dot(d[k:], d[:-k]) / n
        s += 2*w*gamma_k
    return s / n  #variance of the sample mean

def dm_test(loss1, loss2, lag=5):
    """Two-sided DM with NW variance on loss diff."""
    d = np.asarray(loss1) - np.asarray(loss2)
    var_hat = newey_west_var(d, lag=lag)
    if not np.isfinite(var_hat) or var_hat <= 0:
        return np.nan, np.nan
    dm = d.mean() / np.sqrt(var_hat)
    #normal approx for large n
    from math import erf, sqrt
    p = 2 * (1 - 0.5*(1 + erf(abs(dm)/np.sqrt(2))))
    return float(dm), float(p)

#============== DM comparisons: per- , per-token =================
#Update paths here:
paths = {
    "QRF":       "qrf_v2_tuned_preds.csv",      # your final QRF v3 preds
    "LQR":       "lqr_pred_paths_full.csv",              # <-- update
    "LightGBM":  "lgb_extended_preds.csv"              # <-- update
}
```

```
dfs = {}
for name, path in paths.items():
    dfp = pd.read_csv(path, parse_dates=["timestamp"])
    #Standardise quantile column names to qXX if necessary (for all models)
    rename_cols = {}
    for col in dfp.columns:
        if col == 'q5_pred':
            rename_cols[col] = 'q5'
        elif col.startswith('q') and col.endswith('00'):
            rename_cols[col] = f"{col}_pred"
        elif col.startswith('q') and 'pred' not in col and col != 'q5':
            rename_cols[col] = f"{col}_pred"
    dfp = dfp.rename(columns=rename_cols)
    for q in ["05","10","25","50","75","90","95"]:
        col_pred = f"q{q}_pred"
        col = f"q{q}"
        if col_pred in dfp.columns and col not in dfp.columns:
            dfp = dfp.rename(columns={col_pred: col})
    needed = {"token","timestamp","y_true"}.union(TAU2COL.values())
    missing = needed - set(dfp.columns)
    #If 'q5' is missing, fill with NaN so assertion does not fail
    if "q5" in missing:
        dfp["q5"] = np.nan
        missing = needed - set(dfp.columns)
    assert not missing, f"{name}: missing columns {missing}"
    dfs[name] = dfp[["token","timestamp","y_true"] + list(TAU2COL.values())].copy()

#Inner-join on token+timestamp so all models are aligned observation-by-observation
base = dfs["QRF"][["token","timestamp"]].copy()
for name in ["LQR","LightGBM"]:
    base = base.merge(dfs[name][["token","timestamp"]], on=["token","timestamp"], how="inner

#Build aligned frames for each model
aligned = {}
for name, dfp in dfs.items():
    aligned[name] = base.merge(dfp, on=["token","timestamp"], how="left", suffixes=("",""))

#Compute per-token DM for every  (QRF vs LQR / QRF vs LightGBM)
rows = []
lag = 5  # horizon-1 for 72h overlapping returns (6 bars of 12h)
for tau in TAUS:
    qcol = TAU2COL[tau]
    for tok, _ in aligned["QRF"].groupby("token"):
        g = {m: aligned[m][aligned[m]["token"]==tok] for m in aligned}
        #Intersection rows only (should align already)
        y = g["QRF"]["y_true"].to_numpy()
        mask = np.isfinite(y)
        #losses
        L = {}
        for m in aligned:
            q = g[m][qcol].to_numpy()
```

```
            mask &= np.isfinite(q)
        #apply mask
        y = y[mask]
        for m in aligned:
            q = g[m][qcol].to_numpy()[mask]
            L[m] = pinball_loss_vec(y, q, tau)

        if len(y) < 15:  # skip tiny samples
            continue

        #DM: QRF better if DM < 0 (lower loss)
        dm_lqr, p_lqr = dm_test(L["QRF"], L["LQR"], lag=lag)
        dm_lgb, p_lgb = dm_test(L["QRF"], L["LightGBM"], lag=lag)

        rows.append({"token": tok, "tau": tau,
                     "dm_qrf_vs_lqr": dm_lqr, "p_qrf_vs_lqr": p_lqr,
                     "dm_qrf_vs_lgbm": dm_lgb, "p_qrf_vs_lgbm": p_lgb,
                     "n": int(len(y))})

dm_by_token = pd.DataFrame(rows).sort_values(["tau","token"])
dm_by_token.to_csv(RESULTS_DIR/"tbl_dm_by_token.csv", index=False)
print("Saved →", (RESULTS_DIR/"tbl_dm_by_token.csv").resolve())

#Win/Draw/Loss counts per   (  = 0.05)
summ = []
alpha = 0.05
for tau, g in dm_by_token.groupby("tau"):
    def wdl(dm, p):
        if not np.isfinite(dm) or not np.isfinite(p):
            return "draw"
        if p < alpha and dm < 0:  # QRF has lower loss
            return "win"
        if p < alpha and dm > 0:
            return "loss"
        return "draw"
    wdl_lqr  = g.apply(lambda r: wdl(r["dm_qrf_vs_lqr"],  r["p_qrf_vs_lqr"]),  axis=1).value
    wdl_lgbm = g.apply(lambda r: wdl(r["dm_qrf_vs_lgbm"], r["p_qrf_vs_lgbm"]), axis=1).value
    summ.append({
        "tau": tau,
        "QRF_vs_LQR_win":  int(wdl_lqr.get("win",0)),
        "QRF_vs_LQR_draw": int(wdl_lqr.get("draw",0)),
        "QRF_vs_LQR_loss": int(wdl_lqr.get("loss",0)),
        "QRF_vs_LGBM_win":  int(wdl_lgbm.get("win",0)),
        "QRF_vs_LGBM_draw": int(wdl_lgbm.get("draw",0)),
        "QRF_vs_LGBM_loss": int(wdl_lgbm.get("loss",0)),
    })
dm_counts = pd.DataFrame(summ).sort_values("tau")
dm_counts.to_csv(RESULTS_DIR/"tbl_dm_counts.csv", index=False)
dm_counts

#============== Heatmap of DM statistics (QRF vs LightGBM) =====================
```

```
pivot = dm_by_token.pivot(index="token", columns="tau", values="dm_qrf_vs_lgbm")
plt.figure(figsize=(8, max(4, 0.35*len(pivot))))
im = plt.imshow(pivot.values, aspect="auto", cmap="coolwarm", vmin=-3, vmax=3)  # clip arou
plt.colorbar(im, label="DM statistic (QRF - LGBM)")
plt.xticks(range(len(pivot.columns)), [f"{t:.2f}" for t in pivot.columns], rotation=0)
plt.yticks(range(len(pivot.index)), pivot.index)
plt.title("Per-token Diebold-Mariano: QRF vs LightGBM (pinball loss)")
plt.tight_layout()
plt.gcf().savefig("/Users/james/OneDrive/Documents/GitHub/solana-qrf-interval-forecasting/pa
plt.savefig(RESULTS_DIR/"fig_dm_heatmap_qrf_vs_lgbm.png", dpi=160)
plt.close()
print("Saved heatmap →", (RESULTS_DIR/"fig_dm_heatmap_qrf_vs_lgbm.png").resolve())
```

- **Model Confidence Set (MCS)**

```
#Utilites

import numpy as np, pandas as pd
from pathlib import Path

RESULTS_DIR = Path("results"); RESULTS_DIR.mkdir(exist_ok=True)
TAUS = [0.05,0.10,0.25,0.50,0.75,0.90,0.95]
TAU2COL = {0.05:"q5",0.10:"q10",0.25:"q25",0.50:"q50",0.75:"q75",0.90:"q90",0.95:"q95"}

def pinball_loss_vec(y, q, tau):
    diff = y - q
    return np.maximum(tau*diff, (tau-1)*diff)

def moving_block_bootstrap_indices(n, block_len, rng):
    """Return indices for one bootstrap sample of length n using moving blocks."""
    if n <= block_len:
        start = rng.integers(0, max(1, n-1))
        idx = np.arange(start, min(n, start+block_len))
        return np.resize(idx, n)
    starts = rng.integers(0, n - block_len + 1, size=int(np.ceil(n / block_len)))
    idx = np.concatenate([np.arange(s, s+block_len) for s in starts])[:n]
    return idx

def tokenwise_block_resample(panel, block_len, rng):
    """Resample *within each token* to preserve each token's serial dependence."""
    out = []
    for tok, g in panel.groupby("token", sort=False):
        idx = moving_block_bootstrap_indices(len(g), block_len, rng)
        out.append(g.iloc[idx])
    return pd.concat(out, axis=0, ignore_index=True)

def build_aligned_panel(paths):
    """Return a long panel: columns [token,timestamp,model,tau,loss]."""
    dfs = {}
    for name, path in paths.items():
        dfp = pd.read_csv(path, parse_dates=["timestamp"])
```

```
        #Standardise quantile column names to qXX if necessary (for all models)
        rename_cols = {}
        for col in dfp.columns:
            if col == 'q5_pred':
                rename_cols[col] = 'q5'
            elif col.startswith('q') and col.endswith('00'):
                rename_cols[col] = f"{col}_pred"
            elif col.startswith('q') and 'pred' not in col and col != 'q5':
                rename_cols[col] = f"{col}_pred"
        dfp = dfp.rename(columns=rename_cols)
        for q in ["05","10","25","50","75","90","95"]:
            col_pred = f"q{q}_pred"
            col = f"q{q}"
            if col_pred in dfp.columns and col not in dfp.columns:
                dfp = dfp.rename(columns={col_pred: col})
        needed = {"token","timestamp","y_true"}.union(TAU2COL.values())
        missing = needed - set(dfp.columns)
        #If 'q5' is missing, fill with NaN so assertion does not fail
        if "q5" in missing:
            dfp["q5"] = np.nan
            missing = needed - set(dfp.columns)
        assert not missing, f"{name}: missing columns {missing}"
        dfs[name] = dfp[["token","timestamp","y_true"] + list(TAU2COL.values())].copy()

    #Align on the intersection of timestamps per token across all models
    base = dfs[next(iter(dfs))][["token","timestamp"]].copy()
    for name in dfs:
        if name == next(iter(dfs)):
            continue
        base = base.merge(dfs[name][["token","timestamp"]], on=["token","timestamp"], how="i

    panels = []
    for name, dfp in dfs.items():
        g = base.merge(dfp, on=["token","timestamp"], how="left")
        long = []
        for tau, qcol in TAU2COL.items():
            loss = pinball_loss_vec(g["y_true"].to_numpy(), g[qcol].to_numpy(), tau)
            long.append(pd.DataFrame({
                "token": g["token"].values,
                "timestamp": g["timestamp"].values,
                "model": name,
                "tau": tau,
                "loss": loss
            }))
        panels.append(pd.concat(long, axis=0, ignore_index=True))
    panel = pd.concat(panels, axis=0, ignore_index=True)
    #keep finite rows only
    panel = panel[np.isfinite(panel["loss"])].reset_index(drop=True)
    return panel

def mcs_once(loss_mat, models, rng, block_len=6, B=1000, alpha=0.10):
```

```
"""
Hansen et al. MCS using the Tmax statistic:
- d_i,t = l_i,t - mean_j l_j,t
- t_i = sqrt(n)*mean(d_i)/sd_bootstrap(mean(d_i)^*)
- T_max = max_i t_i; eliminate argmax if p < alpha
Returns surviving models and elimination log.
"""
current = list(models)
elim_log = []
#loss_mat: dataframe with columns ['token','timestamp'] + models, for a fixed
base_cols = ["token","timestamp"]
key = loss_mat[base_cols].copy()

while len(current) > 1:
    L = loss_mat[current].to_numpy()
    n = L.shape[0]
    #d_i,t relative to cross-model mean
    d = L - L.mean(axis=1, keepdims=True)  # (n, m)
    dbar = d.mean(axis=0)                  # (m,)
    #bootstrap means of d_i
    dbar_boot = []
    for b in range(B):
        #resample tokenwise with blocks
        boot_idx = []
        for tok, g in loss_mat.groupby("token", sort=False):
            idx = moving_block_bootstrap_indices(len(g), block_len, rng)
            #Map to the corresponding rows of this tau-specific matrix
            start = g.index.min()
            boot_idx.append(start + idx)
        boot_idx = np.concatenate(boot_idx)
        db = d[boot_idx, :].mean(axis=0)
        dbar_boot.append(db)
    dbar_boot = np.vstack(dbar_boot)  # (B, m)
    #studentized t_i
    sd = dbar_boot.std(axis=0, ddof=1)
    #avoid zeros
    sd = np.where(sd <= 1e-12, np.inf, sd)
    t_i = np.sqrt(n) * dbar / sd
    T_obs = np.max(t_i)

    #bootstrap Tmax
    t_i_boot = np.sqrt(n) * (dbar_boot - dbar) / sd
    T_boot = np.max(t_i_boot, axis=1)
    pval = float((T_boot >= T_obs).mean())

    #stop if we can't reject EPA
    if pval >= alpha:
        break

    #eliminate worst model (largest t_i)
    worst_idx = int(np.argmax(t_i))
```

```
        worst_model = current[worst_idx]
        elim_log.append({"eliminated": worst_model, "Tmax": float(T_obs), "pval": pval, "k"
        current.pop(worst_idx)
        #drop the model from loss_mat
        loss_mat = loss_mat.drop(columns=[worst_model])

    return current, pd.DataFrame(elim_log)

#Run MCS across   (pooled over tokens)

#Set your file paths here
paths = {
    "QRF":      "qrf_v2_tuned_preds.csv",
    "LQR":      "lqr_pred_paths_full.csv",     # <-- update to your path
    "LightGBM": "lgb_extended_preds.csv"     # <-- update to your path
}

panel = build_aligned_panel(paths)

rng = np.random.default_rng(42)
alpha = 0.10
B = 1000
block_len = 6  # 6×12h = 72h overlap

survivors, logs = [], []

for tau in TAUS:
    sub = panel[panel["tau"] == tau].copy()
    if sub.empty:
        survivors.append({"tau": tau, "survivors": "no data"})
        continue

    sub = sub.sort_values(["token","timestamp","model"])
    pivot = sub.pivot_table(index=["token","timestamp"], columns="model", values="loss", agg

    #Ensure all model columns exist; if absent, create filled with NaN
    for m in paths.keys():
        if m not in pivot.columns:
            pivot[m] = np.nan

    pivot = pivot.reset_index()

    #Keep only models actually present as columns
    present_models = [m for m in paths.keys() if m in pivot.columns]
    if len(present_models) < 2:
        survivors.append({"tau": tau, "survivors": "insufficient models"})
        continue

    #Drop rows with NaN in any of the present models (so comparisons are aligned)
    pivot = pivot.dropna(subset=present_models)
    if pivot.empty or pivot.shape[0] < 20:
```

```
        survivors.append({"tau": tau, "survivors": "insufficient data"})
        continue

    #If more than 1 model present, run MCS on those
    keep, log = mcs_once(
        loss_mat=pivot[["token","timestamp"] + present_models],
        models=present_models,
        rng=rng, block_len=block_len, B=B, alpha=alpha
    )

    survivors.append({"tau": tau, "survivors": ",".join(keep)})
    if len(log):
        log["tau"] = tau
        logs.append(log)

mcs_survivors = pd.DataFrame(survivors)
mcs_log = pd.concat(logs, ignore_index=True) if len(logs) else pd.DataFrame(columns=["elimi

mcs_survivors.to_csv(RESULTS_DIR/"tbl_mcs_survivors.csv", index=False)
mcs_log.to_csv(RESULTS_DIR/"tbl_mcs_elimination_log.csv", index=False)
print("Saved:", (RESULTS_DIR/"tbl_mcs_survivors.csv").resolve(), (RESULTS_DIR/"tbl_mcs_elim
mcs_survivors
```

- **Reliability by Regime**

```
df = pd.read_csv("features_v1_tail.csv", parse_dates=["timestamp"])
pred = pd.read_csv("qrf_v2_tuned_preds.csv", parse_dates=["timestamp"])
#resolve regime labels for (token,timestamp)
def resolve_regime_labels(df_fold: pd.DataFrame) -> pd.Series:
    if "vol_regime" in df_fold.columns:
        reg = df_fold["vol_regime"]
        if pd.api.types.is_numeric_dtype(reg):
            out = pd.Series(np.where(reg >= 3, "volatile", np.where(reg <= 1, "quiet", "mid"
                            index=reg.index, dtype="object")
            out[reg.isna()] = "mid"
            return out
        m = {"low":"quiet","quiet":"quiet","calm":"quiet","mid":"mid","normal":"mid",
             "high":"volatile","volatile":"volatile","wild":"volatile"}
        return reg.astype(str).str.lower().map(m).fillna("mid")
    #fallback: proxy vol (first available)
    proxy = next((c for c in ["gk_vol_36h","parkinson_vol_36h","vol_std_7bar","downside_vol_
    if proxy:
        v = df[proxy]
        q1, q2 = v.quantile([0.33,0.66])
        out = pd.Series(np.where(v>=q2,"volatile",np.where(v<=q1,"quiet","mid")), index=v.in
        out[v.isna()] = "mid"
        return out
    return pd.Series(["mid"]*len(df), index=df.index, dtype="object")

#Define column names
TOKEN_COL = "token"
```

```
TIME_COL = "timestamp"

reg_labs = df[[TOKEN_COL, TIME_COL]].copy()
reg_labs["regime"] = resolve_regime_labels(df)

#Ensure timestamp columns are datetime for merge
reg_labs[TIME_COL] = pd.to_datetime(reg_labs[TIME_COL])
pred["timestamp"] = pd.to_datetime(pred["timestamp"])

#Join with predictions (filtered set if you want)
pred_reg = pred.merge(reg_labs.rename(columns={TOKEN_COL:"token", TIME_COL:"timestamp"}),
                      on=["token","timestamp"], how="left")

rows=[]
for rg, g in pred_reg.groupby("regime"):
    y = g["y_true"].to_numpy()
    rows.append({
        "regime": rg,
        "coverage80": ((y>=g["q10"])&(y<=g["q90"])).mean(),
        "coverage90": ((y>=g["q5"]) &(y<=g["q95"])).mean(),
        "width80": (g["q90"]-g["q10"]).mean(),
        "width90": (g["q95"]-g["q5"]).mean(),
        "n": len(g)
    })
reg_tbl = pd.DataFrame(rows).sort_values("regime")
reg_tbl.to_csv(RESULTS_DIR/"tbl_reliability_by_regime_qrf.csv", index=False)
reg_tbl
```

- **Sharpness Coverage**

```
#Paths (update if yours are different)
paths = {
    "QRF":      Path("qrf_v2_tuned_preds.csv"),
    "LQR":      Path("lqr_pred_paths_full.csv"),
    "LightGBM": Path("lgb_extended_preds.csv")
}

eff_rows = []
for name, p in paths.items():
    if not p.exists():
        print(f"Warning: missing {name} predictions at {p}, skipping.")
        continue
    d = pd.read_csv(p, parse_dates=["timestamp"])
    #If columns are named 'q5', 'q10', etc., rename to 'q05_pred', 'q10_pred', etc.
    rename_cols = {}
    for col in d.columns:
        if col == 'q5':
            rename_cols[col] = 'q05_pred'
        elif col == 'q10':
            rename_cols[col] = 'q10_pred'
        elif col == 'q90':
```

```
            rename_cols[col] = 'q90_pred'
        elif col == 'q95':
            rename_cols[col] = 'q95_pred'
    if rename_cols:
        d = d.rename(columns=rename_cols)
    need = {"token", "timestamp", "y_true", "q05_pred", "q10_pred", "q90_pred", "q95_pred"}
    if not need.issubset(d.columns):
        print(f"Warning: {name} file missing required columns, skipping.")
        continue
    y = d["y_true"].to_numpy()
    eff_rows += [
        {"model": name, "interval": "80%", "coverage": ((y >= d["q10_pred"]) & (y <= d["q90_
        {"model": name, "interval": "90%", "coverage": ((y >= d["q05_pred"]) & (y <= d["q95_
    ]

eff = pd.DataFrame(eff_rows)
eff.to_csv(RESULTS_DIR / "tbl_efficiency_scatter.csv", index=False)
eff
```

- **Backtesting**

```
#===== Quantile-to-signal backtest (72h horizon, non-overlapping) ==============
import numpy as np, pandas as pd, matplotlib.pyplot as plt
from pathlib import Path

RESULTS = Path("results"); RESULTS.mkdir(exist_ok=True)

#---- 1) Load predictions
def load_preds(path):
    d = pd.read_csv(path, parse_dates=["timestamp"])
    need = {"token","timestamp","y_true","q5","q10","q25","q50","q75","q90","q95"}
    missing = need - set(d.columns)
    assert not missing, f"{path} missing: {missing}"
    d = d.sort_values(["token","timestamp"]).reset_index(drop=True)
    #ensure numeric
    for c in ["y_true","q5","q10","q25","q50","q75","q90","q95"]:
        d[c] = pd.to_numeric(d[c], errors="coerce")
    return d

models = {}
models["QRF"] = load_preds("qrf_v2_tuned_preds.csv")
#Optional peers (comment out if unavailable)
if Path("lgbm_preds.csv").exists():
    models["LightGBM"] = load_preds("lgbm_preds.csv")
if Path("lqr_preds.csv").exists():
    models["LQR"] = load_preds("lqr_preds.csv")

#2) Position sizing policies
EPS = 1e-6
S_MAX = 1.0          # per-token cap (absolute size)
GROSS_CAP = 1.0      # portfolio gross cap at each timestamp (sum |sizes| <= GROSS_CAP)
```

```
#Costs per side in basis points
FEE_BPS  = 15        # e.g., 0.15% per side total (DEX fee + misc)
SLIP_BPS = 10        # slippage estimate per side
COST_PER_SIDE = (FEE_BPS + SLIP_BPS) / 1e4  # convert bps to decimals

MIN_EDGE = 0.0       # optional |q50| filter for Policy B

def size_policy_A(df):
    """Risk-scaled continuous sizing: s = clip(q50 / (|q10|+eps), [-S_MAX,S_MAX])."""
    s = (df["q50"] / (df["q10"].abs() + EPS)).clip(-S_MAX, S_MAX)
    return s

def size_policy_B(df):
    """High-confidence, thresholded: long if q10>0, short if q90<0, else 0. Optional |q50| f
    s = np.where(df["q10"] > 0,  1.0,
        np.where(df["q90"] < 0, -1.0, 0.0))
    s = np.where(np.abs(df["q50"]) >= MIN_EDGE, s, 0.0)
    return pd.Series(s, index=df.index, dtype=float)

def apply_gross_cap(frame, size_col="size", cap=GROSS_CAP):
    """At each timestamp, scale sizes so sum |size| <= cap."""
    g = frame.groupby("timestamp")[size_col].apply(lambda s: np.maximum(s.abs().sum(), 1e-12
    scale = (cap / g).reindex(frame["timestamp"]).to_numpy()
    #only scale where sum|s|>cap
    sumabs = frame.groupby("timestamp")[size_col].transform(lambda s: s.abs().sum())
    need = sumabs > cap + 1e-12
    out = frame[size_col].copy()
    out.loc[need] = out.loc[need] * scale[need.to_numpy()]
    return out

def backtest_on_predictions(pred_df, policy_fn, label):
    df = pred_df.copy()
    #Build raw sizes
    df["size"] = policy_fn(df).astype(float)
    #Cap per-token
    df["size"] = df["size"].clip(-S_MAX, S_MAX)
    #Cap portfolio gross per timestamp
    df["size"] = apply_gross_cap(df, "size", GROSS_CAP)

    #Round-trip cost deducted at entry: 2 * COST_PER_SIDE * |size|
    round_trip_cost = 2.0 * COST_PER_SIDE
    df["ret_gross"] = df["size"] * df["y_true"]
    df["ret_net"]   = df["ret_gross"] - round_trip_cost * df["size"].abs()

    #Aggregate to portfolio (equal across tokens after sizing)
    port = df.groupby("timestamp")["ret_net"].mean().to_frame("ret").reset_index()

    #Metrics on 72h-step series
    r = port["ret"].to_numpy()
    mean = float(np.nanmean(r))
```

```
    std  = float(np.nanstd(r))
    downside = r[r<0]
    sortino = mean / (np.nanstd(downside) + 1e-12) if downside.size else np.nan
    sharpe  = mean / (std + 1e-12)

    #Max drawdown
    nav = (1.0 + port["ret"]).cumprod()
    roll_max = nav.cummax()
    max_dd = float(((nav/roll_max)-1).min())

    #Hit-rate (directional correctness)
    hit = float((np.sign(df["size"]) * df["y_true"] > 0).mean())

    #Turnover proxy (sum |size| per timestamp)
    avg_gross = float(df.groupby("timestamp")["size"].apply(lambda s: s.abs().sum()).mean())

    out = {
        "model": label,
        "policy": policy_fn.__name__,
        "mean_ret": mean,
        "vol": std,
        "sharpe": sharpe,
        "sortino": sortino,
        "max_drawdown": max_dd,
        "hit_rate": hit,
        "avg_gross": avg_gross,
        "periods": len(port),
        "trades": len(df)
    }
    return out, port, df[["token","timestamp","size","y_true","ret_net"]]

#---- 3) Run: QRF with both policies (and peers if you like) -------------------
rows = []
curves = {}
tradelogs = {}

for mname, preds in models.items():
    for policy in (size_policy_A, size_policy_B):
        res, port, trades = backtest_on_predictions(preds, policy, mname)
        key = f"{mname}-{policy.__name__}"
        rows.append(res)
        curves[key] = port.assign(nav=(1+port["ret"]).cumprod())
        trades.to_csv(RESULTS/f"trades_{key}.csv", index=False)

perf = pd.DataFrame(rows).sort_values(["model","policy"]).reset_index(drop=True)
perf.to_csv(RESULTS/"tbl_backtest_perf.csv", index=False)
perf
```

- **Risk-Aware Sizing Backtest**

```
import numpy as np, pandas as pd, matplotlib.pyplot as plt
```

```
from pathlib import Path

pred = pd.read_csv("qrf_v2_tuned_preds.csv", parse_dates=["timestamp"]).sort_values(["token"
EPS = 1e-6
S_MAX = 1.0
GROSS_CAP = 1.0
FEE_BPS, SLIP_BPS = 15, 10
COST_PER_SIDE = (FEE_BPS + SLIP_BPS)/1e4

def policy_A(df):  # risk-scaled continuous
    return (df["q50"] / (df["q10"].abs() + EPS)).clip(-S_MAX, S_MAX)

def policy_B(df):  # thresholded
    return np.where(df["q10"]>0, 1.0, np.where(df["q90"]<0, -1.0, 0.0))

def apply_gross_cap(frame, size_col="size", cap=GROSS_CAP):
    scale = (cap / frame.groupby("timestamp")[size_col].apply(lambda s: max(s.abs().sum(),1e
    scale = scale.reindex(frame["timestamp"]).to_numpy()
    out = frame[size_col].copy()
    need = frame.groupby("timestamp")[size_col].transform(lambda s: s.abs().sum()) > cap + 1
    out.loc[need] = out.loc[need] * scale[need.to_numpy()]
    return out

def token_backtest(df_tok, policy_fn):
    df = df_tok.copy()
    df["size"] = policy_fn(df)
    df["size"] = apply_gross_cap(df, "size", GROSS_CAP)
    rt_cost = 2.0*COST_PER_SIDE
    df["ret_net"] = df["size"]*df["y_true"] - rt_cost*df["size"].abs()
    #72h step equity for this token
    nav = (1+df["ret_net"]).cumprod()
    r = df["ret_net"].to_numpy()
    sharpe = float(np.nanmean(r) / (np.nanstd(r)+1e-12))
    sortino = float(np.nanmean(r) / (np.nanstd(r[r<0])+1e-12)) if np.any(r<0) else np.nan
    max_dd = float(((nav / nav.cummax()) - 1).min())
    hit = float((np.sign(df["size"]) * df["y_true"] > 0).mean())
    return {"sharpe": sharpe, "sortino": sortino, "max_dd": max_dd, "hit": hit}, df.assign(n

#Per-token metrics for both policies
rows=[]
for tok, g in pred.groupby("token"):
    mA, _ = token_backtest(g, policy_A)
    mB, _ = token_backtest(g, policy_B)
    rows.append({"token": tok, "A_sharpe":mA["sharpe"], "A_sortino":mA["sortino"], "A_maxDD"
                            "B_sharpe":mB["sharpe"], "B_sortino":mB["sortino"], "B_maxDD":mB["
tok_perf = pd.DataFrame(rows).sort_values("A_sharpe", ascending=False)
tok_perf.to_csv("results/tbl_token_backtest_qrf.csv", index=False)
tok_perf.head(10)
```

Akyildirim, E., Goncu, A. and Sensoy, A. (2021) 'Prediction of cryptocurrency returns using machine learning', *Annals of Operations Research*, 297(1), pp. 3–36.

Bai, Y. *et al.* (2021) 'Understanding the under-coverage bias in uncertainty estimation', *arXiv preprint arXiv:2104.05818* [Preprint].

Barber, R.F. *et al.* (2021) 'Predictive inference with the jackknife+', *The Annals of Statistics*, 49(1), pp. 486–507.

Bergmeir, C., Hyndman, R.J. and Koo, B. (2018) 'A note on the validity of cross-validation for evaluating autoregressive time series prediction', *Computational Statistics & Data Analysis*, 120, pp. 70–85.

Bollerslev, T. (1986) 'Generalized autoregressive conditional heteroskedasticity', *Journal of Econometrics*, 31(3), pp. 307–327.

Borri, N. (2019) 'Conditional tail-risk in cryptocurrency markets', *Journal of Empirical Finance*, 50, pp. 1–19.

Breiman, L. (2001) 'Random forests', *Machine Learning*, 45(1), pp. 5–32.

Campbell, J.Y., Lo, A.W. and MacKinlay, A.C. (1997) *The econometrics of financial markets*. Princeton, NJ: Princeton University Press.

Catania, L. and Sandholdt, M. (2019) 'Bitcoin at high frequency', *Journal of Risk and Financial Management*, 12(1), p. 36.

Chernozhukov, V., Fernández-Val, I. and Galichon, A. (2010) 'Quantile and probability curves without crossing', *Econometrica*, 78(3), pp. 1093–1125.

Diebold, F.X. and Mariano, R.S. (1995) 'Comparing predictive accuracy', *Journal of Business & Economic Statistics*, 13(3), pp. 253–263.

Diebold, F.X. and Yilmaz, K. (2014) 'On the network topology of variance decompositions: Measuring the connectedness of financial firms', *Journal of Econometrics*, 182(1), pp. 119–134.

Dyhrberg, A.H. (2016) 'Bitcoin, gold and the dollar–a GARCH volatility analysis', *Finance Research Letters*, 16, pp. 85–92.

Easley, D., O'Hara, M. and Basu, S. (2019) 'From mining to markets: The evolution of bitcoin transaction fees', *Journal of Financial Economics*, 134(1), pp. 91–109.

Engle, R.F. and Manganelli, S. (2004) 'CAViaR: Conditional autoregressive value at risk by regression quantiles', *Journal of Business & Economic Statistics*, 22(4), pp. 367–381.

Friedman, J.H. (2001) 'Greedy function approximation: A gradient boosting machine', *The Annals of Statistics*, 29(5), pp. 1189–1232.

Gkillas, K. and Katsiampa, P. (2018) 'An application of extreme value theory to cryptocurrencies', *Economics Letters*, 164, pp. 109–111.

Gneiting, T. and Raftery, A.E. (2007) 'Strictly proper scoring rules, prediction, and estimation', *Journal of the American Statistical Association*, 102(477), pp. 359–378.

Harvey, D.I., Leybourne, S.J. and Newbold, P. (1997) 'Testing the equality of prediction mean squared errors', *International Journal of Forecasting*, 13(2), pp. 281–291.

Ke, G. *et al.* (2017) 'LightGBM: A highly efficient gradient boosting decision tree', in *Advances in neural information processing systems 30*. Curran Associates, Inc., pp. 3146–3154.

Koenker, R. and Bassett, G. (1978) 'Regression quantiles', *Econometrica*, 46(1), pp. 33–50.

Koutmos, D. (2018) 'Return and volatility spillovers among cryptocurrencies', *Economics Letters*, 173, pp. 122–127.

Liu, Y. and Tsyvinski, A. (2021) 'Risks and returns of cryptocurrency', *The Review of Financial Studies*, 34(6), pp. 2689–2727.

Mcnally, S., Roche, J. and Caton, S. (2018) 'Predicting the price of bitcoin using machine learning', in *2018 26th euromicro international conference on parallel, distributed and network-based processing (PDP)*. IEEE, pp. 339–343.

Meinshausen, N. (2006) 'Quantile regression forests', *Journal of Machine Learning Research*, 7, pp. 983–999.

Romano, Y., Patterson, E. and Candès, E.J. (2019) 'Conformalized quantile regression', in *Advances in neural information processing systems 32*. Curran Associates, Inc., pp. 3543–3553.

Sebastião, H. and Godinho, P. (2021) 'Forecasting cryptocurrency prices with blockchain network metrics', *Expert Systems with Applications*, 177, p. 114944.

Taylor, J.W. (2008) 'Using exponentially weighted quantile regression to estimate value at risk and expected shortfall', *Journal of Financial Econometrics*, 6(3), pp. 382–406.

West, K.D. (1996) 'Asymptotic inference about predictive ability', *Econometrica*, 64(5), pp. 1067–1084.