ChatGPT

Blueprint for Forecasting Financial Time Series

1. Series Taxonomy & Cadence

Frequency & Time Zones: Define each series' natural frequency (daily, weekly, monthly) and ensure consistent time boundaries. For daily series, decide on the day boundary (e.g. midnight in a specific time zone) to aggregate intraday data [1]. Weekly aggregations should align to a defined week start (e.g. Monday) [2]. For example, if raw data is hourly but forecasts are daily, first align hours to their calendar day (using a chosen timezone) and sum or average within each day [1]. The forecast frequency dictates how training data is aggregated (daily data can yield weekly forecasts by summing 7 days per week) [3]. Use **aggregation rules** appropriate to the metric: *additive* series (cash flow, sales) use sums over periods; *averaged* metrics use means (weighted if necessary). Ensure currency amounts are consistently in one currency (or convert/normalize if needed).

Cold-Start Handling: Establish rules for minimal history. If a series has very few points (e.g. < N_min), avoid complex models. The fewer the observations, the simpler the model should be [4]. In fact, with <20 points, automatic ARIMA tends to select extremely simple models (often 0–2 parameters) due to estimation uncertainty [4]. Thus: - If data points < 2 seasonal cycles (or < ~30 points for non-seasonal): start with naive or average forecasts (e.g. last value or mean of history) as baseline. - If data is just a handful of points, consider **no forecast** or use external benchmarks (e.g. industry growth rates) until more data accrues. - Use **warm-up periods**: e.g. require at least 12 points for monthly forecasts or 90 days for daily, etc. If below threshold, indicate high uncertainty and possibly widen prediction intervals.

Data Sufficiency Flags: Implement checks to flag if a series has enough data for each model type. For example, require >1 year of daily data to attempt yearly seasonality models. Document and enforce these minimums in the code to prevent unreliable model fitting.

2. Method Selector (Champion/Challenger)

Available Methods: The system should support a range of forecasting techniques, from simple heuristics to advanced statistical models, and automatically choose a suitable champion model while keeping challengers for comparison:

- **Heuristic Methods:** Simple but robust baselines:
- *Last value (Naïve)* – forecast equals the last observed value [5].
- *Seasonal naïve* – forecast equals last known value from the same season (e.g. last week's same-day value for daily series) [6].
- *Last-k average* – mean of the last k observations (or last k same-day-of-week values). For example, a 7-day moving average for daily data [7].

- *Profile-based* – e.g. average by weekday or month: "on Mondays we typically see X" (captures day-of-week or month-of-year patterns by averaging historical values for that weekday/month).

- **Exponential Smoothing (ETS):** This family covers:

- *Simple Exponential Smoothing (SES)* – appropriate for level-only series (no trend/seasonality). Formula: $L_t = \alpha y_t + (1-\alpha)L_{t-1}$, where L_t is the level (smoothed value at time t) [8]. The forecast is constant: $\hat{y}_{t+1|t} = L_t$.
- *Holt's Linear Trend* – captures level + trend. Equations:
    - Level $\ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1})$,
    - Trend $b_t = \beta^(\ell_t - \ell_{t-1}) + (1-\beta^)b_{t-1}$,
    - Forecast $\hat{y}_{t+h|t} = \ell_t + h\,b_t$. (Here α and β^* are smoothing parameters for level and trend) [9].
- *Holt-Winters (Triple ES)* – for level + trend + seasonality. For additive seasonality, the recurrence is:
    - $\ell_t = \alpha (y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1}+b_{t-1})$,
    - $b_t = \beta^(\ell_t - \ell_{t-1}) + (1-\beta^)b_{t-1}$,
    - $s_t = \gamma (y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$,
    - Forecast: $\hat{y}{t+h|t} = \ell_t + h\,b_t + s$ (with seasonal period m) [10]. This model uses three smoothing parameters (α, β*, γ) for level, trend, season. Multiplicative seasonality is similar but scales the seasonal component (useful when seasonal swings grow with the trend) [11] [12].
- *Damped Trend* – Holt's method with a damping factor $0<\phi<1$ to prevent trends from extrapolating unrealistically far. The forecast becomes $\hat{y}_{t+h|t} = \ell_t + (\phi + \phi^2 + \dots + \phi^h)b_t$ [13].

Include formulas in documentation for transparency and to allow QA of implementation. These smoothing methods are fast and often effective for short-term forecasts of many business metrics [5] [6].

- **ARIMA/SARIMA:** Auto-Regressive Integrated Moving Average models, including seasonal ARIMA:
- Use the Hyndman-Khandakar algorithm (unit root tests for differencing, then AICc-based order search) to automatically select p,d,q (and seasonal P,D,Q) [14]. The algorithm tries combinations of AR terms, MA terms, and differences, seeking the model with minimum AICc (a sample-size corrected Akaike criterion) [14]. It also checks for stationarity and invertibility and may enforce heuristics (e.g. avoid excessive parameters for short series).
- *Seasonality detection:* If the series has a clear seasonal period (e.g. weekly seasonality for daily data, or yearly for monthly data) and enough history, allow seasonal terms. Auto-selection can test models with and without seasonal terms and choose via AICc. If the seasonal period is uncertain, consider using autocorrelation or periodogram analysis to identify it.
- After fitting, ensure residuals have no autocorrelation (white noise check) and reasonably normal distribution (for valid prediction intervals).

- **Note:** For very short series, ARIMA may default to trivial models (random walk or noise) due to lack of data [4]. In such cases, simpler methods (or human judgment) prevail.

- **Prophet-Style Decomposition:** Facebook Prophet (now Oracles like *NeuralProphet*) approach:

- Prophet is an additive regression model: $y_t = g(t) + s(t) + h(t) + \epsilon_t$ [15], where:
    - $g(t)$ is a piecewise linear or logistic growth trend (with automatic changepoint detection),
    - $s(t)$ is seasonal effects (e.g. weekly, yearly Fourier series) [16],
    - $h(t)$ are holiday/event effects (user-provided lists of special dates) [15].

- Prophet is robust to outliers and missing days, and it excels with strong seasonal patterns and several seasons of history [17]. It's less suitable for very short or non-seasonal series.

- In this blueprint, a Prophet-like model can be a **challenger** especially for daily series with complex seasonalities (weekly seasonality + yearly trend + holidays). If chosen, log or seasonal multiplicative modes can handle growth in seasonal magnitude [18] [19].

- **TBATS & Multiple Seasonality:** For series with multiple seasonal cycles (e.g. daily data with *weekly* and *annual* seasonality, or hourly data with daily and weekly patterns), consider TBATS:

- TBATS = Trigonometric seasonality, Box-Cox transform, ARMA errors, Trend, Seasonal components [20]. It models seasonal components using Fourier terms (trigonometric) and can handle non-integer seasonality or multiple seasonal periods [21].
- It automatically tries including/excluding components (e.g. damped trend, ARMA errors) and selects by AIC [22].

- This method is computationally heavier, but as a challenger it shines for complex seasonal structures where ARIMA or Holt-Winters might struggle (e.g. a daily series with both weekly and yearly cycles) [21]. Use it when data length is sufficient for both seasonal cycles (usually >2 years of daily data for yearly seasonality to be learned).

- **Uncertainty via Monte Carlo:** Particularly for *cash flow* forecasts (where volatility and risk are concerns), include a Monte Carlo simulation layer:

- Monte Carlo simulation repeatedly samples random variations to produce a distribution of possible future outcomes [23] [24]. For a given point forecast model, you can simulate future errors by either:
    - Sampling from the model's residuals (bootstrapping residuals) or assumed error distribution, and
    - Iteratively forecasting forward: at each step $t+h$, draw a random error $\tilde{\epsilon}*{t+h} $ and set $\tilde{y}}* = \hat{y*{t+h}* + *\tilde{\epsilon}*$ for the next step.}$, then update the model state with $\tilde{y}_{t+h
- Do this many times (e.g. 1,000 paths) to build a distribution of forecasts. This is useful for cash flow to model variability of inflows/outflows.
- The Monte Carlo method acknowledges we cannot pinpoint exact outcomes, so it generates many random trials to approximate the probability distribution of forecasts [25] [26]. It is a powerful way to visualize best/worst case cash scenarios beyond static intervals. Ensure random seed control for reproducibility in production.

Champion/Challenger Framework: Rather than a one-size-fits-all model, use a champion-challenger approach: - **Champion model**: the currently selected best method for the series (based on validation performance). - **Challengers**: other models that are fit in parallel (or periodically) and evaluated on new data. For example, if ARIMA is champion, keep a Holt-Winters model and a Prophet model as challengers running. Monitor their errors; if a challenger consistently outperforms the champion on backtests or out-of-sample, consider "promoting" it. - This approach allows continuous improvement and guards against concept drift. DataRobot's MLOps, for instance, endorses champion/challenger deployments where a challenger can replace the champion if it proves superior [27] [28]. No single model is best forever [29], so we periodically **re-evaluate**.

Auto-Selection Logic: Implement a decision flow (either code logic or a flowchart in documentation) such as:

- **Step 1: Data characteristics.** Check series length and seasonality.
- If length > one year of daily data (or > 2 seasonal periods in general) and clear seasonal pattern → allow seasonal models (Seasonal ARIMA, Holt-Winters, Prophet, TBATS).
- If length is moderate (enough for trend but maybe <1 season) → focus on non-seasonal models (ARIMA without season, Holt linear).
- If very short series or highly irregular → use heuristic or simpler exponential smoothing (SES).
- If series is nearly constant or intermittent (many zeros) → use naive or Croston's method (for intermittency) as a specialized case.
- **Step 2: Try ARIMA first (if applicable).** ARIMA is often a strong baseline for many series [30]. Use auto-ARIMA to fit the best ARIMA/SARIMA. If the series shows significant seasonal spikes in ACF and you have enough data, include seasonal order search. If ARIMA model fit is successful (residuals okay) and the in-sample/validation error is reasonably low, tentatively select ARIMA.
- If ARIMA fails to fit (e.g. due to singularities or lack of data) or yields high error, note that and consider alternate methods.
- **Step 3: Try Exponential Smoothing models.** If ARIMA wasn't suitable or as a challenger:
- If seasonality identified → fit Holt-Winters (additive or multiplicative based on whether seasonal amplitude grows with level [11] [12]).
- If no seasonality but trend → fit Holt's linear (possibly with damping).
- If no trend/season → fit SES.
- Record the validation metrics.
- **Step 4: Heuristic baseline.** Always compute a baseline forecast (naive or seasonal naive) for reference. Many forecasting workflows compare against naive because sometimes simple methods are surprisingly hard to beat [5] [31]. Ensure this baseline error is computed to avoid overly complex models that barely outperform a naive forecast.
- **Step 5: Prophet/TBATS (if applicable).** For daily/weekly data with complex patterns or holiday effects, fit a Prophet model on the training set including known holidays and regressors. Also, if multiple seasonalities present (e.g. daily transactions with both weekly and yearly cycles), fit a TBATS model. These can be resource-intensive, so perhaps only run if simpler models aren't clearly sufficient or if user specifically requests higher granularity in seasonal effects.
- **Step 6: Compare and select.** Using a validation scheme (see Backtesting & Metrics), compare the models' performance (MAPE, RMSE, etc.). Also consider **practicality**: a slightly higher error model may be preferable if it's much simpler or faster, unless the accuracy difference is material. Typically:
- If ARIMA and Holt-Winters perform similarly, prefer ARIMA if interpretability of components is needed or HW if stability/simplicity is desired.

- If Prophet adds value by incorporating holidays that ARIMA/ETS missed (e.g. big holiday spikes), it might win for those cases.

- **Fallback ladder:** If ARIMA is feasible and yields good metrics, choose it as champion (Holt-Winters as fallback if ARIMA fails diagnostic tests or data is too sparse for ARIMA). If both ARIMA and Holt-Winters are unsuitable (e.g. very short series or nonstationary issues), fall back to heuristic methods (average or naive). Document this hierarchy clearly.

- **Model Registry Output:** Once chosen, register the champion model details:

- model_id (unique identifier for this model/series run),
- model_type (e.g. “ARIMA”, “Holt-Winters-additive”),
- model_parameters (e.g. ARIMA(1,1,0)(0,1,1)[12] or smoothing α,β,γ values),
- training_data_range (e.g. 2019-01-01 to 2024-12-31 used for training),
- detected_frequency/seasonality (daily with m=7 weekly seasonality, etc.),
- features_used (e.g. “holiday_US=true, payday_schedule=true” if exogenous features included).

This registry can be a database table or JSON file (see *Data Contracts* below) that allows traceability of what was deployed. It’s critical for auditability and for warm-starting (reusing model states).

3. Feature Engineering

Enhance the base time series with additional features to help models capture patterns:

- **Calendar Features:** Derive time-based features:
- Day-of-week (1–7) for daily data to capture weekday vs weekend effects.
- Day-of-month, end-of-month indicator (for series with monthly billing spikes or salary payments on month-end).
- Week-of-year or month-of-year for yearly seasonal patterns in high-frequency data.

- **Periodic profiles:** For example, create a “Monday” dummy or separate average for each weekday to inform heuristic profiles or as regressors in an ARIMAX/Prophet model.

- **Special Dates & Holidays:** Incorporate known holidays or events that impact the metrics:

- Create binary flags for major holidays (national holidays, Black Friday/Cyber Monday for retail sales, etc.). Prophet allows adding holiday effects which essentially allocate a coefficient for those days [15] [32] .
- If different regions/users have different holiday schedules, use the user’s locale to generate relevant holiday features.

- Mark period-ends like quarter-end or year-end if those cause spikes (common in finance: e.g. end of quarter reporting or budget flush in spend data).

- **Promotions/Events:** If there are known promotional events (sales campaigns, product launches) that affect product-level sales, include them as features. This could be a flag or a magnitude (e.g. percent discount offered). They act as external regressors to help explain unusual peaks.

- **Seasonal Decomposition Features:** If not using models that handle seasonality inherently, you can compute seasonal indices from history (e.g. average sales on each weekday normalized) and use those as features or adjust the series by removing seasonal mean before modeling (and adding it back to forecasts).

- **Lag Features / Autoregressive terms:** For machine-learning approaches (if any) or to enrich regression models, you might include lagged values (yesterday's value, last week's same-day value, etc.) as features. However, in pure statistical forecasting, the ARIMA/ETS inherently use past values so explicit lag features are usually not needed unless doing an ML model.

- **Inflation & Currency Normalization:** For financial series measured in currency, provide option to normalize for inflation or exchange rate:

- If forecasting in real terms, adjust historical values by CPI or relevant price index to current currency value, forecast in real terms, then re-inflate the forecast.
- If users have data in different currencies, convert them to a base currency using FX rates if cross-series comparison is needed. Document clearly if forecasts are "in nominal USD" or "in 2025 USD" etc.

- This is especially relevant for long-term forecasts (e.g. multi-year MRR projections) where inflation can materially change values.

- **Outlier Detection & Treatment:** Identify anomalous points in the history that could skew model training:

- *Detection:* Implement a procedure to flag outliers, e.g. any point that is beyond say 3σ from a moving average, or using statistical tests on residuals from a preliminary model. There are functions (like `tsoutliers` in R) that detect outliers by fitting an ARIMA and finding large residuals.
- *Correction:* Once identified, decide on handling: You can replace outliers with a more typical value (e.g. the median of neighbors or interpolated value) **for model-fitting purposes**, while storing the actual value separately. Alternatively, include an indicator feature for that timestamp so that the model knows something unusual happened.
- For example, a one-time large transaction inflating daily cash flow could be capped or removed when training (and perhaps forecasted separately if needed).

- **Missing data**: Similarly, fill missing timestamps (forward-fill or treat as zeros if that makes sense, or use interpolation) consistently so that models, especially those requiring regular frequency, can run.

- **Feature Utilities:** Provide a library of transformations that can be toggled via `forecast_specs`. E.g., in the `forecast_specs` for a series, one could list `features_used: ["dow", "holiday_US", "promo_flag", "outlier_adjustment"]` indicating which feature engineering steps were applied for transparency.

- **Exogenous Regressors:** Apart from calendar, allow user-provided exogenous data. For instance, if forecasting product sales, an exogenous might be Google search trends for that product, or economic indicators for net cash flow (like unemployment rate or consumer spending index). ARIMAX and Prophet can include such regressors [33] [34]. Be mindful to **align** these series in time and **have future values** available (for forecast horizon) or you'll need to forecast the exogenous inputs too.

By enriching the input space, we make our models more informed. However, keep models as simple as possible – add features only if you expect a pattern cannot be learned from the raw time series alone. Each added feature increases complexity and potential for overfitting, so use domain knowledge to pick the most relevant ones (e.g. paydays and holidays are known drivers for personal cash flow and retail spend).

4. Backtesting & Metrics

To ensure models will perform well on future data, use backtesting (evaluating forecasts on historical holdout samples):

Rolling-Origin Evaluation: Utilize a rolling-origin (time series cross-validation) approach [35] [36]. This means we simulate how we would have forecasted in the past at multiple points: - **Expanding Window:** Start with an initial training period (e.g. first 2 years), forecast the next h (forecast horizon, say 3 months), record the errors; then expand the training set to include that next period, forecast again, etc. The training window grows with each origin (we always use all data up to the origin) [37]. - **Sliding Window:** Alternatively, keep the training window size fixed (especially if concerned about structural changes or to limit computation) and roll it forward in time. For example, always use the past 3 years of data to forecast the next 1 month, then move ahead. - The rolling origin method preserves temporal order (no lookahead) and provides multiple forecast-error samples, which is more reliable for estimating out-of-sample accuracy [35].

Backtest Strategy: Define how far back and how many splits: - For daily series, you might use the last 90 days as test, forecasting 7 days ahead in a rolling manner (12 week-long forecast simulations). - For monthly series, if you have 3 years data, perhaps use the last year as test, forecasting 1-3 months ahead in each split. - Ensure the test set covers at least one season or a few cycles of the business to get a representative error. - Automate this: the system should allow a "backtest" mode where it iteratively trains and forecasts for each origin, collecting errors.

Evaluation Metrics: Use a suite of metrics as each has pros/cons [38] [39]: - **MAE (Mean Absolute Error):** Easy to interpret (average absolute error in same units as data) [38]. - **RMSE (Root Mean Squared Error):** Penalizes larger errors more due to squaring; useful if big errors are disproportionately bad [38]. RMSE is more sensitive to outliers than MAE. - **MAPE (Mean Absolute Percentage Error):** Average percentage error ${\frac{100}{n}\sum |y-\hat{y}|/y}$ [39]. Easy to understand as "on average, forecast is off by X%". However, it can be skewed or undefined if actuals are zero or very low [40]. Use with caution if zeros present (e.g. retention rate series might have 0% sometimes). - **SMAPE (Symmetric MAPE):** Uses $(|y-\hat{y}|)/((|y|+|\hat{y}|)/2)$ form [41]. It's bounded between 0–200% and treats positive/negative errors more evenly [41]. Still can behave oddly near zero, and literature suggests it has issues [42], but it's often required in competitions or by stakeholders familiar with it. - **WAPE or WMAPE (Weighted MAPE):** Essentially $\frac{\sum |y-\hat{y}|}{\sum y}$ [43]. This weights errors by actual volume, which is very useful in financial contexts (e.g. a \$100 error on a \$1000 day is more important than on a \$10,000 day, and WAPE reflects that). WAPE is the total absolute error divided by total actual, sometimes expressed as a percentage. It handles zeros better (as long as the sum of actuals over the period isn't zero). - **MASE (Mean Absolute Scaled Error):** Scales errors relative to a naive baseline [44] [45]. If MASE < 1, your model is better than a naive forecast on the training set [46]. This is a good metric for comparing across series because it's scale-free and benchmarked. - **Coverage of Prediction Intervals:** For probabilistic forecasts, track how often the actual falls within certain prediction intervals. For example, over the backtest runs, calculate the fraction of times the actual value fell below the forecast P90 upper bound and above the P10 lower bound. Ideally: - ~80% of actuals should fall between P10 and P90 if those are 10th–90th percentile interval forecasts. - ~50% inside P25–P75 if using those, etc. This "coverage" metric directly measures calibration of uncertainty [47] [48]. For instance, if your 90% interval is too narrow, actual coverage might be only 70%; if too wide, coverage might be 100% (intervals always contain actual, but maybe too broad to be useful). - **Interval Width/Sharpness:** Along with coverage, note the average width of intervals (P90–P10 range). A model with proper coverage but narrower intervals is better (more informative) [49]. So we aim for high coverage *and* small width – there's a trade-off [50].

Model Selection Criteria: Use the above metrics on backtests to choose champion models. For example: - Prioritize **MAPE/WAPE** for business communication (easy percent interpretation) but double-check with MAE/RMSE for absolute error magnitude. - If one model has significantly lower MAPE and WAPE than others, that's a strong sign. If metrics disagree (one model better in RMSE, another in MAPE), analyze why (could be one model is better on large-volume days vs small-volume days). - Also consider *prediction interval accuracy*: if a model consistently miscalibrates uncertainty, that might be a drawback if the use-case needs risk estimates. For instance, if an ARIMA's 90% interval often captures only 80% of actuals, it underestimates variability [47]. - **Operational considerations:** If two models are close in accuracy, prefer the simpler or more stable one. E.g., Holt-Winters might be chosen over ARIMA if they tie, since it's faster and easier to update online.

Model Promotion Rules: Establish policies for when to retrain or switch models: - If the champion's error in recent forecasts exceeds a threshold (say MAPE doubles compared to validation) and a challenger model is performing better on the same recent period, consider promoting the challenger. - Schedule periodic re-selection (e.g. every quarter, re-run backtests on latest data for all candidate methods). - Always retain history of model changes for governance. When a model is replaced, log the date, reason (e.g. "SMAPE of ARIMA exceeded that of Holt-Winters by 5 percentage points in last 3 months"), and the new champion's details.

Retraining Frequency: It depends on how fast the underlying patterns change: - For daily series that are volatile (like cash flow), you might retrain the model every day or week to incorporate newest data. - For monthly series like MRR, retraining monthly might suffice (since one new data point per month). - In general, implement retraining as a configurable cadence (and possibly event-driven: retrain if new data deviates significantly from forecasts). - Heavyweight models (Prophet with holidays, TBATS) might be retrained less frequently due to cost, unless needed.

Finally, package the backtesting results into the **forecast_metrics** data contract (described later) so you can track how each model did in simulation. This will feed into reporting dashboards or alerting systems to tell analysts how much confidence to have in the forecasts.

5. Uncertainty & Intervals

Point forecasts are incomplete without an expression of uncertainty. This blueprint incorporates **prediction intervals** and distributions:

Analytical vs Bootstrapped Intervals: - Many statistical models provide *analytical formulas* for prediction intervals under assumptions (usually normal errors). For example, ARIMA gives a forecast variance that grows with horizon, and one can compute 80% or 95% intervals as $\hat{y}*t \pm z*_t)$. Exponential Smoothing models also have closed-form intervals in state space form. These intervals are fast to compute but rely on model assumptions (e.g. normality, correct model specification). - }\cdot \mathrm{SE}(\hat{y**Bootstrapped intervals:** An alternative is to empirically simulate intervals by resampling residuals (or using *simulation-based* approaches). As mentioned, bootstrapping the residuals and generating many forecast paths can yield prediction intervals that are more robust, especially if residuals are not perfectly normal [51]. Bootstrapped intervals often perform better when the model's assumptions (like normal distribution of errors) are violated [52]. For instance, they can capture skew or fat-tails in the forecast distribution by using the actual residual distribution rather than a normal approximation [52]. - **Example:** For an ARIMA, you could take the in-sample residuals, center them, and on each simulation step add a randomly sampled residual (with replacement) instead of a normal random error. Do 1000 simulations to build an empirical distribution for each forecast horizon.

Monte Carlo Path Generation: (As described in Method Selector, Monte Carlo is also used for model selection in our system for cash flow). Using the final model, generate $N$ sample paths of length $H$ (forecast horizon):

```
function simulate_forecast_paths(model, N, H):
    outputs = matrix(N x H)
    for i in 1..N:
        model_copy = clone(model)  # copy to avoid affecting original
        for h in 1..H:
            point, sd = model_copy.forecast(1)  # one-step ahead forecast &
stddev
            # Sample error: if model provides sd, assume normal; or bootstrap
from residuals:
            if analytic:
                 err = Normal(0, sd)
            else:
                 err = sample(residuals)  # bootstrap
            y_sim = point + err
            outputs[i, h] = y_sim
            model_copy.update(y_sim)  # incorporate as if this happened
    return outputs
```

After this, we have (for each horizon step $h$) a distribution of $N$ simulated values. We then compute quantiles: e.g. $p_{90}(h)$ = 90th percentile of ${ outputs[1..N, h] }$, similarly $p_{10}, p_{50}$ etc. These give us our $P90/P10$ interval and median forecast.

- For implementation, note that some libraries (e.g. `forecast` in R) have built-in simulate functions [53] that handle the model state. Use those if available to avoid reinventing the wheel.

Prediction Interval Coverage: We emphasize that the interpretation of (say) a 90% interval is: in **future use**, 90% of actual future points should fall inside this band [54]. During backtesting, measure if this holds. If not, calibrate: - If coverage is too low (actual falls outside too often), the intervals are too narrow (underestimation of uncertainty). You might inflate them by using a higher percentile of residuals or adding a safety margin. - If coverage is too high (intervals rarely breached, e.g. 100% coverage when you expected 90%), intervals may be too wide (overestimation). You can narrow them or check if some outliers inflated the variance.

Storing Distributions: In our data contract (forecast_outputs), we plan to store a few key quantiles (P10, P50, P90). This is usually sufficient for most reporting (e.g. “best case / median / worst case”). If full distribution is needed (for scenario analysis), one could store a serialized distribution or a set of simulation sample points. However, that can be heavy. We opt for quantiles unless otherwise required. The structure might be:

```
{ "series_key": "...", "date": "...", "point": X, "p10": L, "p50": M, "p90":
U, ... }
```

where *point* could be mean or median forecast (we will use median = P50 as point forecast for symmetry, unless users prefer mean).

For measures like *net cash flow* which might feed into risk calculations, a Monte Carlo histogram could be stored or at least metrics like Probability of Negative Cash (which can be derived from distribution).

Bootstrapping vs. Analytical – Recommendation: - By default, use analytical intervals for speed (especially for ARIMA/ETS which provide them). They give a quick read. - Validate them by comparing to a bootstrap interval on the backtest. If there’s a big discrepancy (due to non-normal residuals or model bias), consider switching to bootstrapped intervals in production or at least flagging that uncertainty is higher than analytic formula suggests. - In high-stakes forecasting (e.g. financial planning), the extra compute for Monte Carlo is usually worth it for a better grasp of tail risks.

To summarize, our system will output not just a single forecast line but a **forecast distribution** per period. This enables users to plan for various scenarios (pessimistic, optimistic) rather than a single guess. The *forecast_specs* will indicate if intervals are analytic or simulation-based, and *forecast_metrics* will include interval calibration info (e.g. “P90 historical coverage = 88%” as a check).

6. Operationalization

Design the forecasting pipeline to run reliably in a production (embedded) environment:

Scheduling Cadence: Determine how often forecasts are generated for each series type: - **Daily series (e.g. net cash flow, product daily sales):** Likely run forecast generation daily. For net cash flow, you might forecast 30 days ahead every morning. Product daily sales might be forecasted each day for the next 14 days, for instance. The schedule could be every night after close of business, new data is processed and forecast updated. - **Monthly series (category spend, MRR):** Run these after each month’s data is finalized. For example, on the 1st of each month, run forecasts for the next 12 months of MRR. A mid-month run could also be scheduled if needed for updated outlook, but usually monthly data forecasts are updated monthly. - **Weekly series (if any):** could be every week. - These schedules should be configurable per series or per client. Use a job scheduler or cron with appropriate timezone awareness (especially if users in different time zones — ensure that daily jobs align with end-of-day in the user’s locale if needed).

Batch vs Real-time: Likely, we schedule batch forecasts, but we may also allow on-demand forecasting (e.g. a user opens their financial app and requests a latest forecast). The system should support both without inconsistency: - For on-demand, ensure it uses the same logic and doesn't conflict with scheduled runs (perhaps by writing with unique run identifiers or upserting results).

Warm-Starting Models: Idempotency and efficiency: - Many models (ARIMA, ETS) can be updated with new data rather than refit from scratch. For example, an ARIMA model can take the last state (last values of AR terms, last residuals) and update coefficients with one more data point (Kalman filter updates) rather than re-estimate everything. In practice, though, re-estimating with auto.arima regularly might be simpler but at cost of CPU. - Provide capability to warm-start: e.g., store the last model object at T and when new data $T+1$ arrives, either: - Use `model.update(new_obs)` if library supports (statsmodels has extend method, R forecast has Arima() on new data with fixed params or state update). - Or at least use previous parameters as initial values for the next optimization (to speed convergence). - If model structure might change (auto-ARIMA might choose a different order with more data), warm-starting the same model may not be optimal in the long run. Thus, consider periodically doing a full re-selection despite warm-starting in between if needed. - **Holt-Winters warm-start:** straightforward: just feed the new data into the recurrence once to update level/trend/season components. - Document which models are warm-started vs retrained fully. Perhaps: - ARIMA: retrain monthly with full search, but in daily incremental runs, just extend (if performance acceptable). - ETS: can be extended one step without recomputation (since smoothing equations can just be applied for the new point). - Prophet: does not easily warm-start (it's not recursive in that way), so likely retrain Prophet models when needed (but Prophet is relatively fast for moderate data). - Monte Carlo simulation: not applicable (always rerun for forecast).

Idempotent Runs: Ensure running the forecast process twice for the same period doesn't double-save or produce different results: - Use a **composite key** for forecast outputs such as (user_id, series_key, forecast_date, run_timestamp). If the same job runs twice for the same effective date, either update the existing records or have a clear versioning. - One strategy: include `computed_at` timestamp and ensure only the latest is considered active. Or maintain a separate table of active forecast vs old forecasts. - If upstream data hasn't changed, the forecast should ideally reproduce the same results (stochastic elements should be controlled by a random seed for consistency, or average out simulation variability by using large N or same seed). - If multiple workers run in parallel, have a locking mechanism keyed by series to avoid race conditions (e.g., use database locks or job queue exclusivity so you don't trigger two forecasts for the same series simultaneously). - Idempotency also means if no new data, you might choose not to produce a new identical forecast (unless you want to just refresh timestamps). This depends on design; some systems store forecasts for each day they run even if identical.

Multi-Tenancy: If this is an embedded solution for many users (as implied by userId/series), ensure one tenant's data doesn't bleed into another's model. Partition data by user, and key all outputs by user. You might run each user's forecast in isolation or batch all users but filter by user.

Resource Management: Some methods like TBATS or large Prophet models are heavy. Possibly schedule those at off-peak hours or with lower frequency. Lighter methods (like ARIMA, ETS) can run more frequently.

Monitoring & Alerts: As part of operationalization, set up monitoring: - If a forecast job fails (e.g. model fitting error, or no data available), log and alert. - If inputs are anomalous (e.g. sudden huge spike in data), perhaps alert an analyst as it might indicate data issues or a real change needing model update. - If forecast outputs are drastically different from previous run (e.g. cash flow forecast changed by 50% in a day), flag it.

In sum, operationalize such that forecasts are produced on schedule, reliably, and are traceable by keys. Each run's outputs link to the model spec used. This is crucial for debugging ("Tuesday's forecast looks odd – which model/params were used?" can be answered via the stored registry).

7. Data Contracts (Projections Schemas)

Define structured schemas for storing forecast configs, outputs, and metrics, using SQL tables or JSON:

1. `forecast_specs` **– Forecast Specification Registry:** This table stores the blueprint of each forecast run or model. Each row identifies *what method was used with what settings* for a specific series (and user, if multi-tenant). For example:

Column	Type	Description
user_id	int	(If multi-tenant) The user or entity for whom the forecast is made.
series_key	text	Identifier for the time series (e.g. "net_cash_flow", "sales_product_123").
model_id	text	Unique ID of the model run (could concatenate user, series, date or use a UUID).
method	text	Model type used (e.g. `"ARIMA"`, `"HOLT-WINTERS(additive)"`, `"PROPHET"`).
parameters	json	Hyperparameters or model order. e.g. `{"order": [1,1,0], "seasonal_order": [0,1,1,12]}` for ARIMA, or smoothing α/β/γ for ETS, etc.
training_start	date	Start date of training data window.
training_end	date	End date of training data window.
seasonal_period	text	Seasonal frequency used, if any (e.g. `"7 days"`, `"12 months"`). Could be null if none.
features_used	text	Comma-separated or JSON list of feature engineering steps or exogenous variables included (e.g. `["dow","holidays_US"]`).
error_metrics	json	(Optional) Store summary of validation metrics for this model (e.g. `{"MAPE_val": 12.3, "MAE_val": 2000}`) – though detailed metrics go in `forecast_metrics` table.
created_at	timestamp	When this spec entry was created (when model was trained).

Example JSON entry (for documentation purposes):

```
{
  "user_id": 42,
  "series_key": "category_spend_marketing",
  "model_id": "42_marketing_2025Q1",
  "method": "SARIMA",
  "parameters": {"order": [0,1,1], "seasonal_order": [0,1,1,12]},
  "training_start": "2020-01-01",
  "training_end": "2024-12-01",
  "seasonal_period": "12 months",
  "features_used": ["month_of_year", "promo_flag"],
  "created_at": "2025-01-02T00:00:00Z"
}
```

This tells us user 42's marketing spend forecast for Q1 2025 was made with a SARIMA(0,1,1)(0,1,1)[12] using monthly seasonality and promo flags as extra features.

2. `forecast_outputs` **– Forecast Results:** This table holds the actual forecasted values for each horizon and quantiles. Each row is one forecast data point (or we can use wide format with all quantiles in one row). Likely a narrow format per date is easiest for querying. Columns:

Column	Type	Description
user_id	int	User identifier (to partition data).
series_key	text	Time series key (matches the one in specs).
model_id	text	Reference to `forecast_specs.model_id` used to produce this forecast.
target_date	date	The date (or date-time) of the forecasted period (what period the forecast is for).
horizon	int	The step-ahead index of this forecast (e.g. 1 for first period ahead, 2 for second, etc.). Could be deduced from comparing target_date and last training date, but stored for convenience.
point_forecast	numeric	The point forecast value (mean or median of distribution).
p10	numeric	10th percentile forecast (lower bound).
p50	numeric	50th percentile (median, which may equal point_forecast if we choose median as point).
p90	numeric	90th percentile (upper bound).
generated_at	timestamp	When the forecast was generated (should match spec's created_at for consistency, but if one spec generates multiple horizons rows, they'll share the timestamp).

Note: The `model_id` ties each forecast to the parameters used. If a new run happens, it will usually create a new model_id and new forecast entries, unless you version differently. If storing forecasts for every run, you might keep old ones for history (with different generated_at). If only the latest is kept, you'd upsert by series_key & target_date.

Example rows: (assuming user_id 42, series “net_cash_flow” daily, generated on 2025-09-26, forecasting 5 days ahead)

```
user_id | series_key    | model_id         | target_date | horizon |
point_forecast | p10    | p50    | p90    | generated_at
------- | ------------- | ---------------- | ----------- | ------- |
-------------- | ------ | ------ | ------ | ---------------------
42      | net_cash_flow | 42_cash_20250926 | 2025-09-27  | 1       |
10500          | 8000   | 10500  | 13000  | 2025-09-26 04:00:00
42      | net_cash_flow | 42_cash_20250926 | 2025-09-28  | 2       |
9800           | 7000   | 9800   | 12500  | 2025-09-26 04:00:00
... (through horizon 5)
```

This shows, for user 42’s cash flow, on Sept 26 we forecast Sept 27’s cash to be 10.5k (with 80% interval [8k, 13k]), Sept 28 to be 9.8k, etc.

3. `forecast_metrics` **– Backtest Metrics:** This table logs how each model performed in backtesting or validation, and potentially live monitoring. It can be structured by model or by model+window. One approach:

Column	Type	Description
user_id	int	User id (if applicable).
series_key	text	Series id.
model_id	text	Reference to model (which should map to `forecast_specs`).
window_start	date	Start of the test window used for evaluation.
window_end	date	End of the test window.
horizon	text	What forecast horizon this evaluation covers (e.g. "1 step ahead", "1-3 ahead", or "all"). Sometimes we evaluate different horizons separately. Could also be null meaning overall.
MAE	numeric	Mean absolute error on this window.
MAPE	numeric	MAPE (%) on this window (if applicable).
SMAPE	numeric	SMAPE (%) on this window.
RMSE	numeric	RMSE on this window.
WAPE	numeric	Weighted absolute % error on this window.
MASE	numeric	Mean scaled error (if baseline is defined).

Column	Type	Description
coverage_p90	numeric	Actual coverage of 90% prediction interval in this window (e.g. 0.85 if 85% of actuals fell within P90 interval).
coverage_p50	numeric	Actual coverage of 50% interval.
… (any other metrics or percentile coverages needed) …		
evaluated_at	timestamp	When this evaluation was done (likely same as model training time if backtest was during training).

Each model could have multiple rows if we do rolling-origin (like one row per fold). Or we could aggregate all folds in one row (e.g. window_start = earliest test, window_end = last test, metrics averaged). It might be useful to store some distribution (like a list of errors) but that’s overkill; we stick to aggregates.

Example:

```
user_id | series_key     | model_id         | window_start | window_end   |
horizon    | MAPE   | RMSE    | WAPE   | coverage_p90 | evaluated_at
42      | net_cash_flow  | 42_cash_20250926 | 2025-06-01   | 2025-08-31   | 1-7
day     | 12.5 | 1500.0 | 0.11 | 0.92         | 2025-09-26 04:00:00
42      | net_cash_flow  | 42_cash_20250926 | 2025-06-01   | 2025-08-31   | 7-14
day    | 18.0 | 2000.0 | 0.15 | 0.85         | 2025-09-26 04:00:00
```

This indicates for model 42_cash_20250926, on the backtest (June–Aug ’25), for 1-week ahead forecasts it got MAPE 12.5%, 90% interval coverage 92%, and for two-weeks ahead horizon MAPE was 18%.

If using expanding window cross-val, we might aggregate overall. Or list multiple time slices if needed. The key is to provide transparency on expected error rates and interval calibration.

Storage and Access: These contracts could be physical tables in an SQL database (for easy querying and joining), and/or JSON objects produced by the forecasting engine stored in a blob store or delivered via API. The structure given above can be translated to a JSON schema if needed.

For example, an API response for a forecast request might look like:

```
{
  "spec": {
    "user_id": 42, "series_key": "net_cash_flow", "model_id":
"42_cash_20250926",
    "method": "ARIMA", "parameters": {"order":[1,0,1],"seasonal_order":[0,0,0,
0]},
    "training_start": "2025-03-01", "training_end": "2025-09-25",
    "features_used": ["dow","holidays_US"], "created_at": "2025-09-26T04:00:00Z"
  },
  "forecast": [
    {"date": "2025-09-26", "point": 10000, "p10": 8000, "p50":10000, "p90":
12000},
    {"date": "2025-09-27", "point": 9500,  "p10": 7600, "p50":9500,  "p90":
11500},
    ...
  ],
  "metrics": {
    "MAPE": 0.125, "WAPE": 0.11, "RMSE": 1500,
    "coverage_p90": 0.92, "coverage_p50": 0.55,
    "evaluated_horizon": "7 days", "evaluation_period": "2025-06-01 to
2025-08-31"
  }
}
```

This shows a possible JSON payload containing spec, forecast, and summary metrics. In practice, the UI can then display "We are 90% confident your cash flow in 7 days will be between X and Y (median Z). Historical MAPE ~12%."

8. Deliverables

Finally, outline the key deliverables of this blueprint implementation:

- **Method Chooser Flowchart:** A diagram (or descriptive flow as above) showing how the system decides on a model. This should illustrate the fallback logic:
- Check data length and seasonality.
- If long & seasonal → try ARIMA/SARIMA.
- If that fails or not adequate → try Holt-Winters (seasonal ETS).
- If still no good → use heuristic (seasonal naive or mean).
- Else if no seasonality detected:
    - Try ARIMA (non-seasonal) or Holt's if trend.
    - If fails → use SES or naive.
- Always compare to baseline; finalize champion.

The flowchart can be delivered as a PDF/image in documentation, ensuring engineers and stakeholders understand model selection. (This flow can be coded as well in the training pipeline.)

- **Pseudocode Implementations:**
- Provide well-commented pseudocode or simplified code for core algorithms:
    - *ETS (Holt-Winters)*: Show the update equations and how forecasting works in a loop (as given in section 2) [10], perhaps including initialization step (like how to get initial level/trend/season from first season of data).
    - *ARIMA (auto-selection)*: Pseudocode should outline the search over p,d,q:

```
best_aicc = inf
for p in 0..Pmax:
  for q in 0..Qmax:
    for d in {0,1,2} (if allowed):
      try model = ARIMA(p,d,q)
      if model fits:
         compute AICc
         if AICc < best_aicc:
             best_model = model
             best_aicc = AICc
```

and include steps for seasonal part if relevant (or mention using library function). Also mention using tests like KPSS or PP to decide differencing automatically [14] .

- *Monte Carlo simulation:* Provide a code snippet (like we did in section 5) to simulate paths and derive quantiles. This pseudocode should highlight generating random errors and updating the model state in a loop.

These pseudocode blocks will be part of the technical documentation or appended in an engineering spec, to guide implementation.

- **SQL & JSON Schemas:** Formal definition of the three data contract tables (`forecast_specs` , `forecast_outputs` , `forecast_metrics`) as given in section 7. This will be delivered as:
- SQL DDL statements to create the tables (with data types appropriate to the DB, e.g. NUMERIC for amounts, JSONB for parameters in Postgres, etc.).

- JSON schema definitions (if using an API), or examples of JSON as above for each. This ensures that the database team or API team can prepare storage and interfaces for the forecast data.

- **Test Plan with Synthetic Scenarios:** We will create at least 20 synthetic time series to rigorously test the forecasting pipeline. Each synthetic series will have known characteristics so we can verify the method selection and forecast accuracy:

- *Pure sinusoidal* (e.g. $\sin(2\pi t/12)$ + noise) – tests if seasonality is detected (should pick seasonal model).
- *Linear trend upward* (with no season, just noise around a line) – test if Holt's trend method or ARIMA with $d=1$ is used.
- *Linear trend with seasonality* (e.g. linear growth + yearly cycle) – expect Holt-Winters or ARIMA with seasonal diff to be chosen.
- *Step change (changepoint) in level* – e.g. series is constant then jumps – test how models react (Prophet can detect changepoint, ARIMA might need $d=1$ to handle it). Evaluate if backtest catches the error around change.
- *Seasonal pattern with trend change* (e.g. seasonal for a while, then level shifts or trend slope changes) – test if models adapt or if champion-challenger might switch (maybe Prophet could handle trend change better).
- *Noisy flat series* (basically white noise) – naive or mean should be chosen, since anything complex would overfit. Ensure our selection logic indeed picks a simple model.
- *AR(1) process* (auto-regressive series) – see if ARIMA picks that up (should choose AR(1) model).

- *MA(1) process* – test ARIMA selection of MA.
- *Seasonal ARIMA process* (e.g. AR(1) with seasonality) – ensure seasonal ARIMA selection works.
- *Intermittent demand series* (mostly zeros, occasional spikes) – test if our pipeline can handle (perhaps flag that specialized method or at least use naive).
- *Multiple seasonalities* (simulate something like TBATS scenario: e.g. daily data with weekly and annual patterns) – check if TBATS is invoked and how it performs vs ignoring one cycle.
- *Multiplicative seasonality* (season amplitude grows with level – e.g. simulate y = trend * (1 + 0.3*sin(…))) – test if multiplicative Holt-Winters or log-transform improves accuracy.
- *Outlier injection* (take a smooth series but add one huge outlier point) – verify outlier detection flags it and model doesn't get thrown off (e.g. ARIMA residual outlier should be caught).
- *Long seasonal period with not much data* (e.g. 18 months of monthly data – has seasonality but only 1.5 cycles) – see if method falls back to simpler (maybe it shouldn't use full seasonal model yet; check cold-start logic).
- *Short series (n<10)* – ensure it just outputs a simple forecast and possibly a warning of uncertainty. The test passes if it doesn't crash and chooses the heuristic.
- *Sudden drop to zero* (e.g. series is trending then goes to zero and flat – like churn) – test if models adapt quickly (maybe require differencing).
- *Trend that inverts (up then down)* – e.g. a quadratic-like series – see if any model can adapt (likely needing piecewise or just will have large errors – but test to ensure no instability).
- *Very high noise-to-signal* (random walk) – see if forecast just ends up basically flat with wide intervals (which is correct behavior – e.g. ARIMA might treat as random walk, produce flat forecast with large variance).
- *Periodic spikes* (e.g. mostly flat but every 6th period a spike – like promo every 6 weeks) – test if feature engineering (promo flag) can help; without it, forecast might smear average; with a known flag, model could predict spikes.
- *Mixture of patterns* (e.g. a seasonal series that also has a trend and an occasional outlier) – a comprehensive test of the full pipeline's robustness.

For each scenario, we will document expected outcome (which model should be picked, how the forecast should behave) and verify the system's output against it.

- **Guidance for 20 Real-World Datasets:** In addition to synthetic tests, we outline using real data to validate performance:
- Collect (or use existing) anonymized financial time series such as:
    1. Daily net cash flow from a sample of small businesses (inflows/outflows).
    2. Monthly spend by category from a personal finance app (e.g. monthly groceries, rent, etc. over 3+ years).
    3. Product-level daily sales from an e-commerce dataset (anonymize product and scale).
    4. Subscription MRR trajectory for multiple products (with events like customer churn or upgrades annotated).
    5. Weekly sales for a retail store (to test weekly frequency).
    6. etc., ensuring a diverse set (some growing, some seasonal decline, some with big holiday spikes like retail in December).

We will run our forecasting pipeline on these real series. The deliverable is a report or notebook showing: - Plots of the forecasts vs actuals (with prediction intervals) to visually inspect reasonableness. - Error metrics computed for these datasets (so stakeholders see actual performance on data similar to their use-case). - If any model consistently underperforms on real data, we adjust the blueprint accordingly. - Guidance will also include how to preprocess these real datasets (e.g. handling missing months in MRR, removing obvious data glitches) in line with our feature engineering and outlier policy.

The final guide will include all the above components. It will serve as a comprehensive reference for engineers implementing the solution and for stakeholders to understand the approach. By adhering to **simple, robust methods** (ETS/ARIMA) first and layering in more complex ones (Prophet, TBATS) as needed, we ensure the system is maintainable and effective with minimal overfitting. All assumptions (like minimum data needed, chosen error metrics) are documented so the blueprint can be adjusted as new information or data characteristics emerge.

---

[1] [2] [3] Data aggregation for different forecast frequencies - Amazon Forecast
https://docs.aws.amazon.com/forecast/latest/dg/data-aggregation.html

[4] 12.7 Very long and very short time series | Forecasting: Principles and Practice (2nd ed)
https://otexts.com/fpp2/long-short-ts.html

[5] [6] [7] [8] Simple Forecasting Methods. Forecasting | by Ritu Santra | Medium
https://medium.com/@ritusantra/simple-forecasting-methods-627648b86d63

[9] [10] [13] 7.3 Holt-Winters' seasonal method | Forecasting: Principles and Practice (2nd ed)
https://otexts.com/fpp2/holt-winters.html

[11] [12] [18] [19] Multiplicative Seasonality | Prophet
http://facebook.github.io/prophet/docs/multiplicative_seasonality.html

[14] Automatic ARIMA model selection - Cross Validated
https://stats.stackexchange.com/questions/665093/automatic-arima-model-selection

[15] [16] [17] [32] [33] [34] 12.2 Prophet model | Forecasting: Principles and Practice (3rd ed)
https://otexts.com/fpp3/prophet.html

[20] [21] [22] TBATS — sktime documentation
https://www.sktime.net/en/latest/api_reference/auto_generated/sktime.forecasting.tbats.TBATS.html

[23] [24] [25] [26] Monte Carlo Simulation: What It Is, How It Works, History, 4 Key Steps
https://www.investopedia.com/terms/m/montecarlosimulation.asp

[27] Introducing MLOps Champion/Challenger Models | DataRobot Blog
https://www.datarobot.com/blog/introducing-mlops-champion-challenger-models/

[28] Challengers tab - DataRobot docs
https://docs.datarobot.com/en/docs/mlops/monitor/challengers.html

[29] There are no Champions in Long-Term Time Series Forecasting
https://arxiv.org/html/2502.14045v1

[30] [51] [52] Time Series Forecasting: Prediction Intervals | by Brendan Artley | TDS Archive | Medium
https://medium.com/data-science/time-series-forecasting-prediction-intervals-360b1bf4b085

[31] [38] [39] [40] [41] [42] [44] [45] [46] 3.4 Evaluating forecast accuracy | Forecasting: Principles and Practice (2nd ed)
https://otexts.com/fpp2/accuracy.html

35 [PDF] Forecast Evaluation for Data Scientists: Common Pitfalls and Best ...
https://arxiv.org/pdf/2203.10716

36 Forecast evaluation for data scientists: common pitfalls and best ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC9718476/

37 Time Series Cross-Validation: An Essential Technique for Predictive ...
https://medium.com/the-modern-scientist/time-series-cross-validation-an-essential-technique-for-predictive-modeling-in-time-dependent-data-444693429eea

43 Time Series Evaluation Metrics — MAPE vs WMAPE vs SMAPE — which one to use, why and when? — Part1 | by Vinit Kothari | Medium
https://medium.com/@vinitkothari.24/time-series-evaluation-metrics-mape-vs-wmape-vs-smape-which-one-to-use-why-and-when-part1-32d3852b4779

47 48 49 54 Conformal Prediction for Forecast Prediction Intervals | BIP xTech
https://medium.com/bip-xtech/conformal-prediction-for-forecast-confidence-intervals-427887aa8f14

50 Forecast accuracy metric that involves prediction intervals
https://stats.stackexchange.com/questions/194660/forecast-accuracy-metric-that-involves-prediction-intervals

53 r - simulate.Arima function from the forecast package - Stack Overflow
https://stackoverflow.com/questions/14195782/simulate-arima-function-from-the-forecast-package