# PROBABILISTIC TIME SERIES FORECASTING WITH NUMPYRO: A PRACTICAL GUIDE

JUAN CAMILO ORDUZ

Abstract. This article presents a comprehensive guide to probabilistic time series forecasting using NumPyro, a lightweight probabilistic programming library built on JAX. We demonstrate a progression from simple exponential smoothing models to sophisticated hybrid deep state-space models, emphasizing practical implementation patterns and scalability considerations. Through concrete examples, we illustrate key concepts including the `scan` function for recursive relationships, hierarchical model structures for sharing information across time series, custom likelihoods for censored and intermittent demand data, and model calibration techniques using additional likelihood terms. Two showcase applications—availability-constrained TSB models and Hilbert Space Gaussian Process calibration—highlight the flexibility of probabilistic programming for encoding domain knowledge directly into forecasting models. The integration with the broader JAX ecosystem, including Optax for optimization and Flax for neural network components, enables scalable inference via stochastic variational inference (SVI) on thousands of time series.

## 1. Introduction

Time series forecasting is fundamental to decision-making across industries, from supply chain optimization to energy demand prediction. While point forecasts provide single estimates of future values, probabilistic forecasting goes further by quantifying uncertainty—a critical capability for risk assessment and robust decision-making.

NumPyro [14] is a lightweight probabilistic programming library that provides a NumPy backend for Pyro [2]. It relies on JAX [4] for automatic differentiation and JIT compilation to GPU/CPU, enabling efficient inference on large-scale problems. This combination of expressiveness and performance makes NumPyro particularly well-suited for time series forecasting applications.

**Why Probabilistic Forecasting?** The advantages of probabilistic forecasting extend across multiple dimensions:

- **Interpretability**: Explicit model structure with clear assumptions enables trust in results and facilitates communication with stakeholders.
- **Uncertainty quantification**: Full predictive distributions enable risk assessment and inform decision-making under uncertainty.
- **Customization**: Probabilistic programming frameworks allow encoding domain knowledge through custom likelihoods, priors, and model structures.

*Date*: January 17, 2026.

- **Scale**: Modern inference algorithms combined with GPU acceleration enable fitting models to thousands of time series efficiently.

**Industry Applications.** Probabilistic forecasting finds applications across diverse domains:

- **Supply chain forecasting**: Demand forecasting with stockout constraints, availability considerations, and inventory optimization for long-tail products.
- **Energy forecasting**: Weather-dependent electricity demand modeling with peak load prediction and uncertainty quantification.
- **Price elasticity modeling**: Hierarchical models for SKU-level pricing decisions and promotional effectiveness measurement.
- **Marketing mix modeling**: Measuring media effectiveness with lift test calibration and scenario planning for marketing spend optimization.

**Article Overview.** This article presents a progression from simple to complex forecasting models:

(1) We begin with the `scan` function, the foundational building block for time series models in NumPyro.
(2) We then introduce exponential smoothing models, demonstrating the basic pattern for state-space models.
(3) ARIMA and VAR models extend this framework to autoregressive structures.
(4) Hierarchical models enable sharing information across related time series.
(5) Specialized models address intermittent demand, censored observations, and price elasticity estimation.
(6) Two showcase sections demonstrate advanced techniques: availability-constrained forecasting and model calibration with Hilbert Space Gaussian Processes.
(7) We conclude with stochastic variational inference for scalable inference and hybrid deep state-space models that integrate neural network components.

## 2. Literature Review and Related Work

Probabilistic forecasting has a rich history spanning classical statistical methods and modern deep learning approaches. This section surveys key developments and positions the NumPyro-based approach within this landscape.

2.1. **Classical Statistical Methods. State space models and ETS**: Hyndman et al. [9, 8] reformulated exponential smoothing as innovations state space models, enabling likelihood estimation and automatic model selection. The ETS (Error-Trend-Seasonality) taxonomy provides a systematic framework for exponential smoothing variants.

**ARIMA family**: The Box-Jenkins methodology established autoregressive integrated moving average models as a cornerstone of time series analysis. Extensions include seasonal ARIMA (SARIMA) and vector autoregressive (VAR) models for multivariate forecasting.

**Structural time series**: Harvey [7] and Durbin & Koopman [6] developed comprehensive frameworks for state space modeling with Kalman filtering, providing the theoretical foundation for many modern approaches.

2.2. **Deep Learning Approaches. DeepAR** [17]: A seminal work in probabilistic deep learning for forecasting, DeepAR trains a global autoregressive RNN model across many related time series. The key innovation is learning to share patterns across series while producing probabilistic forecasts through parametric output distributions.

**Temporal Fusion Transformers** [10]: Attention-based architectures for multi-horizon forecasting with interpretable attention weights and handling of static and dynamic covariates.

**N-BEATS and N-HiTS** [13]: Pure deep learning approaches without time series-specific components, using basis expansion for interpretability.

2.3. **Probabilistic Programming Frameworks. Pyro Forecasting Module**: Built on Pyro/PyTorch, this module provides hierarchical models with global-local structure and SVI for scalable inference. The Dynamic Linear Model tutorial [16] demonstrates powerful calibration techniques that we extend in this work.

**Prophet** [21]: Facebook's decomposable model combining trend, seasonality, and holiday effects, designed for business forecasting at scale with Stan backend for uncertainty quantification.

**GluonTS** [1]: A unified Python interface for probabilistic time series modeling, implementing DeepAR, Transformers, and many other methods.

2.4. **Recent Advances. LGT/SGT Models** [18]: Bayesian exponential smoothing with flexible trend components between linear and exponential, using Student-t errors for robustness.

**ADAM** [19]: The Augmented Dynamic Adaptive Model provides a comprehensive state space framework with extensive treatment of intermittent demand and censored observations.

2.5. **Gap Addressed by This Work.** Existing tools often lack easy customization of model components, direct integration of domain knowledge, and transparent model specification. NumPyro addresses these gaps by providing:

- Composable model building with `scan` for time series
- Custom likelihoods (censored, zero-inflated, availability-constrained)
- JAX ecosystem integration for GPU acceleration
- SVI for scalable inference on thousands of series
- Neural network integration via Flax NNX

## 3. Technical Foundation: The Scan Function

The `scan` function is the fundamental building block for time series models in NumPyro. It provides an efficient implementation of sequential computations, essential for models with recursive relationships of the form $y_t \mapsto y_{t+1}$.

Listing 1. Pure Python implementation of the scan function

```
def scan(f, init, xs):
    """Pure Python implementation of scan.

    Parameters
```

```
    ----------
    f : callable
        A function to be scanned: (carry, x) -> (carry, y)
    init : any
        Initial loop carry value
    xs : array
        Values over which to scan along leading axis
    """
    carry = init
    ys = []
    for x in xs:
        carry, y = f(carry, x)
        ys.append(y)
    return carry, np.stack(ys)
```

The transition function `f` takes the current carry state and input, returning an updated carry and output. This pattern naturally maps to state-space model formulations where the carry represents latent states (level, trend, seasonality) and the output represents predictions.

**Remark 1.** In JAX, `jax.lax.scan` compiles the loop efficiently, avoiding Python overhead and enabling automatic differentiation through the entire sequence. NumPyro's `contrib.control_flow.scan` extends this with proper handling of probabilistic primitives.

## 4. Exponential Smoothing Models

Exponential smoothing provides a natural starting point for probabilistic forecasting with NumPyro. We begin with simple exponential smoothing and progressively add complexity.

### 4.1. Simple Exponential Smoothing.

The level equations for simple exponential smoothing are:

$$\begin{aligned} \hat{y}_{t+h|t} &= l_t \\ l_t &= \alpha y_t + (1-\alpha) l_{t-1} \end{aligned}$$

where $y_t$ is the observed value, $l_t$ is the level, and $\alpha \in (0, 1)$ is the smoothing parameter.

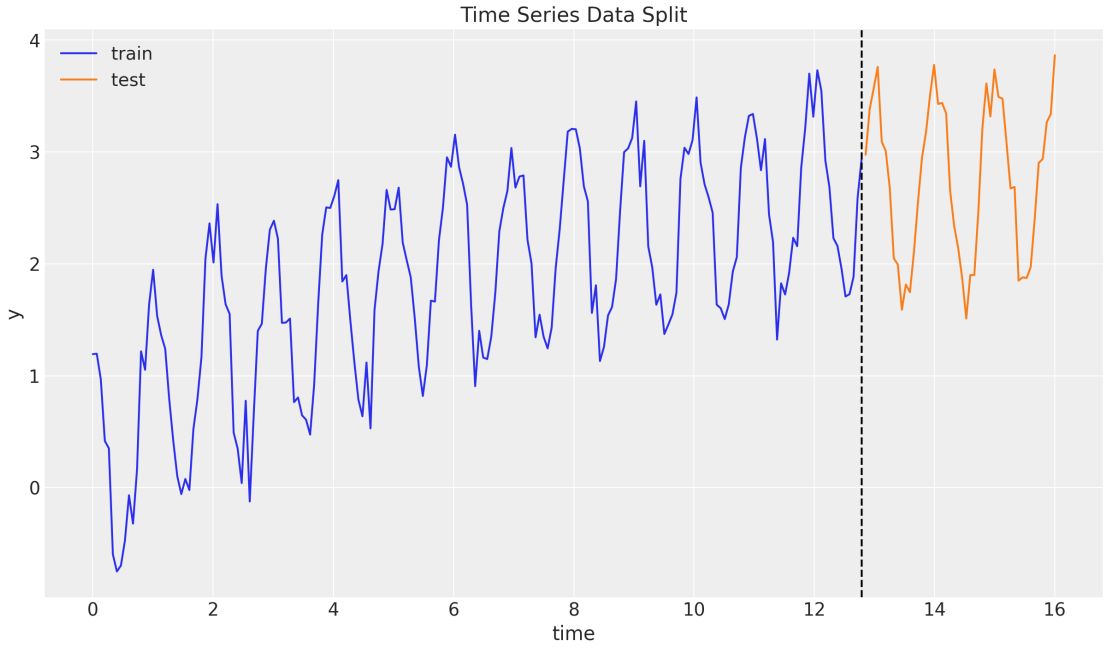


FIGURE 1. Example time series data for exponential smoothing. The data exhibits a clear trend component that simple exponential smoothing must capture through the level dynamics.

The NumPyro implementation uses the `scan` pattern:

LISTING 2. Transition function for simple exponential smoothing

```
def transition_fn(carry, t):
    previous_level = carry

    level = jnp.where(
        t < t_max,
        level_smoothing * y[t] + (1 - level_smoothing) * previous_level,
        previous_level, # Forecast: no update
    )

    mu = previous_level
    pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))

    return level, pred
```

The full model specifies priors for the smoothing parameter, initial level, and observation noise:

LISTING 3. Complete simple exponential smoothing model

```
def level_model(y: ArrayLike, future: int = 0) -> None:
```

```
    t_max = y.shape[0]

    # Priors
    level_smoothing = numpyro.sample(
        "level_smoothing", dist.Beta(concentration1=1, concentration0=1)
    )
    level_init = numpyro.sample("level_init", dist.Normal(loc=0, scale=1))
    noise = numpyro.sample("noise", dist.HalfNormal(scale=1))

    def transition_fn(carry, t):
        # ... as above ...

    # Run scan with conditioning
    with numpyro.handlers.condition(data={"pred": y}):
        _, preds = scan(transition_fn, level_init, jnp.arange(t_max + future))

    if future > 0:
        numpyro.deterministic("y_forecast", preds[-future:])
```

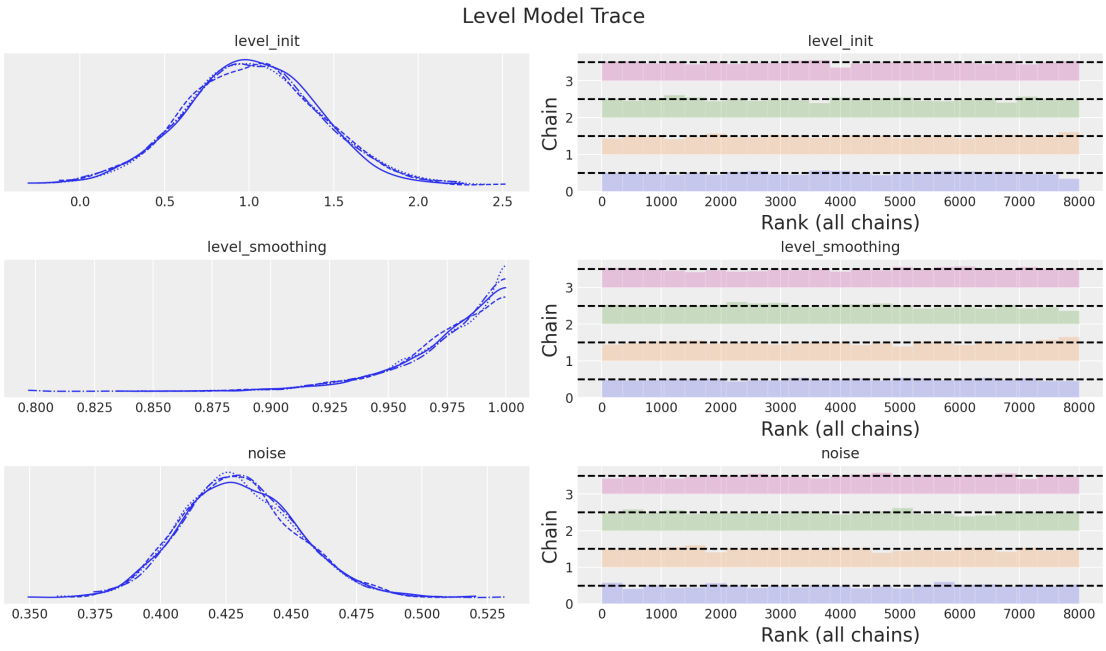


FIGURE 2. Posterior distributions of model parameters. The smoothing parameter $\alpha$ concentrates around 0.8, indicating relatively fast adaptation to recent observations.

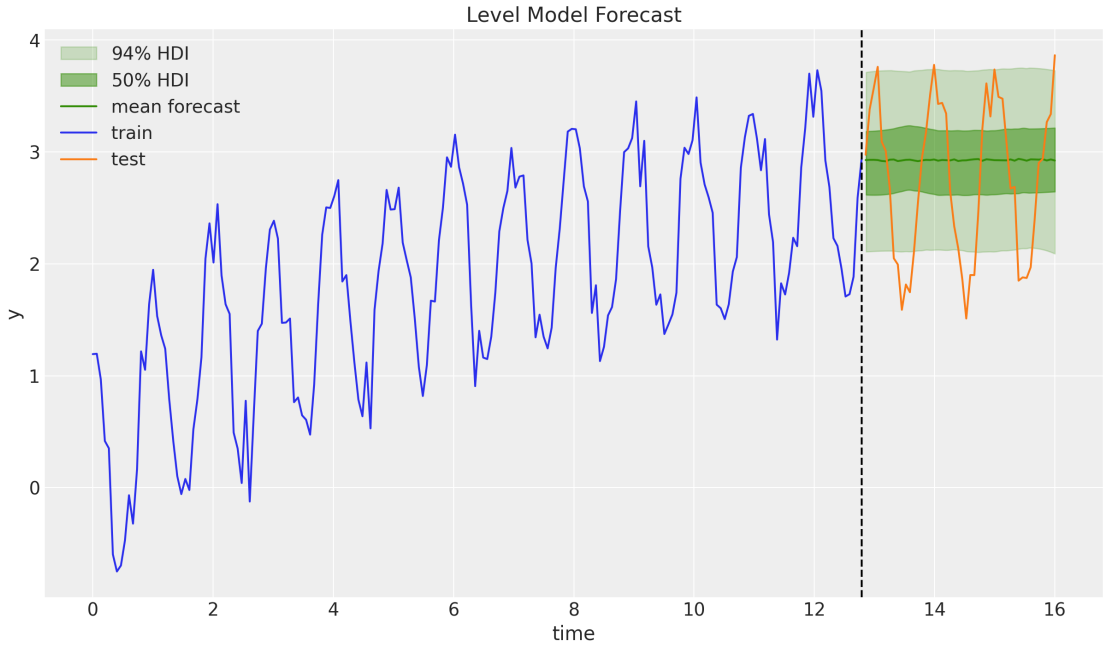


FIGURE 3. In-sample fit and forecast with 94% highest density intervals. The model captures the underlying trend while appropriately quantifying forecast uncertainty.

4.2. **Extensions: Trend, Seasonality, and Damping.** The exponential smoothing framework extends naturally to include trend and seasonal components. The Holt-Winters model with damped trend combines:

$$\begin{aligned}
\hat{y}_{t+h|t} &= l_t + (\phi + \phi^2 + \cdots + \phi^h) b_t + s_{t+h-m(k+1)} \\
l_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + \phi b_{t-1}) \\
b_t &= \beta(l_t - l_{t-1}) + (1-\beta)\phi b_{t-1} \\
s_t &= \gamma(y_t - l_{t-1} - \phi b_{t-1}) + (1-\gamma) s_{t-m}
\end{aligned}$$

where $b_t$ is the trend, $s_t$ is the seasonal component, $\phi$ is the damping factor, and $m$ is the seasonal period.

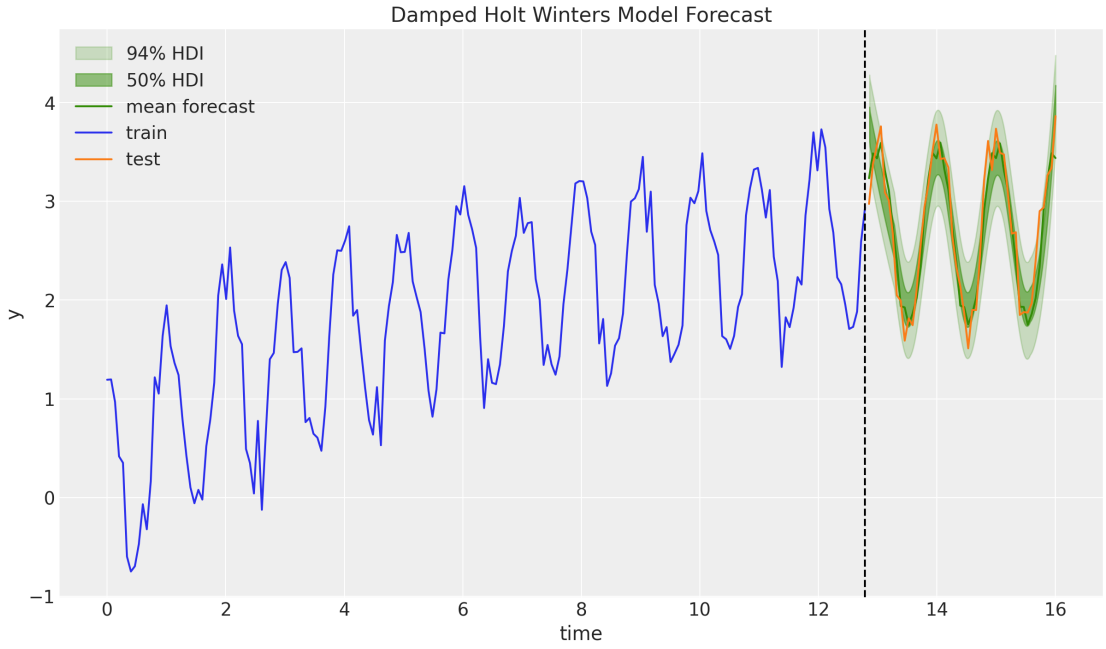


FIGURE 4. Damped trend seasonal model forecast. The model captures both the trend decay and seasonal patterns, with uncertainty bands widening appropriately into the forecast horizon.

## 5. ARIMA MODELS

Autoregressive integrated moving average (ARIMA) models provide an alternative framework for time series modeling. The NumPyro implementation follows the same `scan` pattern.

5.1. **ARMA(1,1) Model.** The ARMA(1,1) model combines autoregressive and moving average components:

$$y_t = \mu + \phi y_{t-1} + \theta \varepsilon_{t-1} + \varepsilon_t$$

LISTING 4. Transition function for ARMA(1,1) model

```
def transition_fn(carry, t):
    y_prev, error_prev = carry
    ar_part = phi * y_prev
    ma_part = theta * error_prev
    pred = mu + ar_part + ma_part
    error = y[t] - pred
    return (y[t], error), error
```

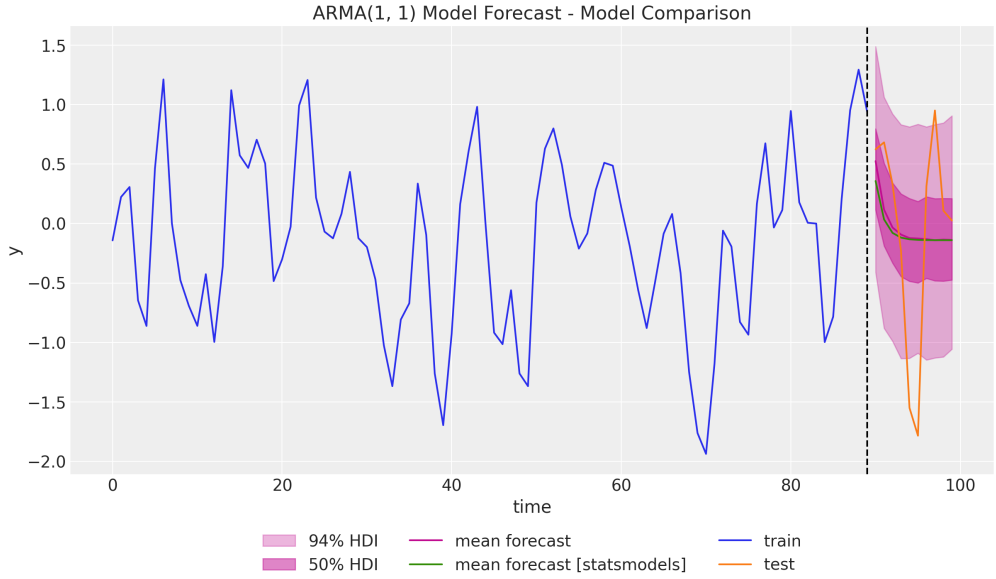


FIGURE 5. ARMA(1,1) model fit showing the in-sample predictions and residuals. The model effectively captures the short-term autocorrelation structure in the data.

5.2. **VAR Models and Impulse Response Functions.** Vector autoregressive (VAR) models extend ARMA to multivariate settings, enabling analysis of dynamic relationships between multiple time series. A key application is computing impulse response functions (IRFs) that trace the effect of a shock to one variable through the system.

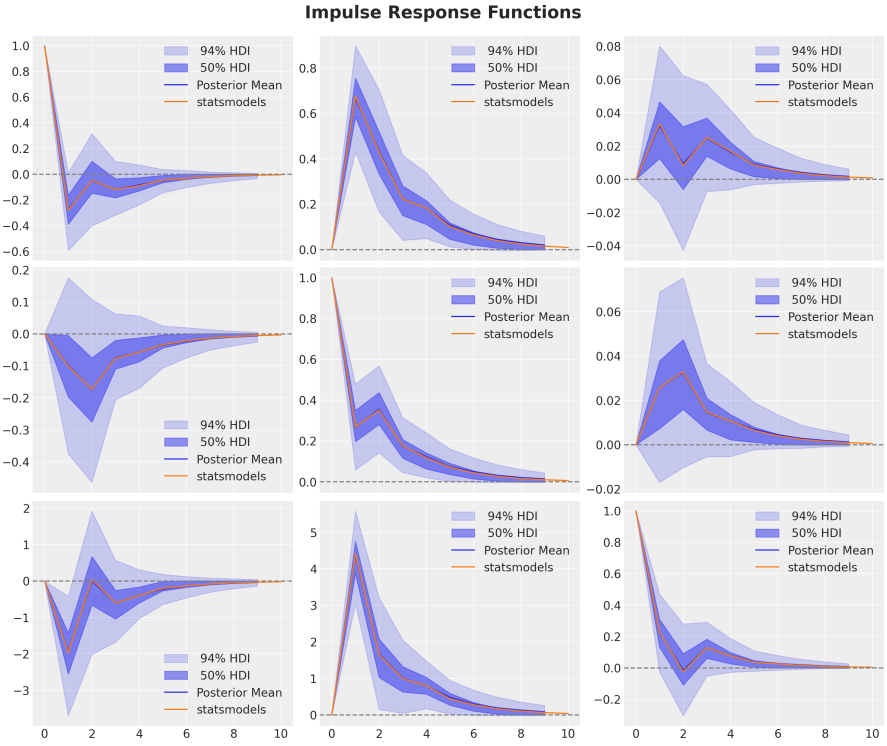


FIGURE 6. Impulse response functions from a Bayesian VAR model. The posterior distributions over IRFs quantify uncertainty in the dynamic relationships between variables.

## 6. HIERARCHICAL MODELS

When forecasting multiple related time series, hierarchical models enable sharing information across series while respecting individual heterogeneity. This approach is particularly valuable when some series have limited historical data.

### 6.1. **Hierarchical Exponential Smoothing.**

We demonstrate hierarchical modeling using Australian tourism data, where visitor counts are organized by region and purpose of travel.


NUTS Forecast
Northern Territory::Darwin::Holiday
Queensland::Darling Downs::Other
Queensland::Outback::Business
Northern Territory::Lasseter::Other
Northern Territory::East Coast::Other
Northern Territory::Hobart and the South::Holiday
South Australia::Adelaide Hills::Business
Victoria::Central Highlands::Visiting
Victoria::Central Murray::Other
Victoria::Ballarat::Holiday
trips
train
test
HoltWinters forecast
train-test split
94% HDI
50% HDI
posterior predictive mean

The hierarchical structure places priors on the smoothing parameters that allow information sharing:

$$\begin{aligned} \alpha_i &\sim \text{Beta}(\mu_\alpha \kappa, (1-\mu_\alpha)\kappa) \\ \mu_\alpha &\sim \text{Beta}(2, 2) \\ \kappa &\sim \text{HalfNormal}(10) \end{aligned}$$

where $\mu_\alpha$ is the population mean smoothing parameter and $\kappa$ controls the concentration around this mean.

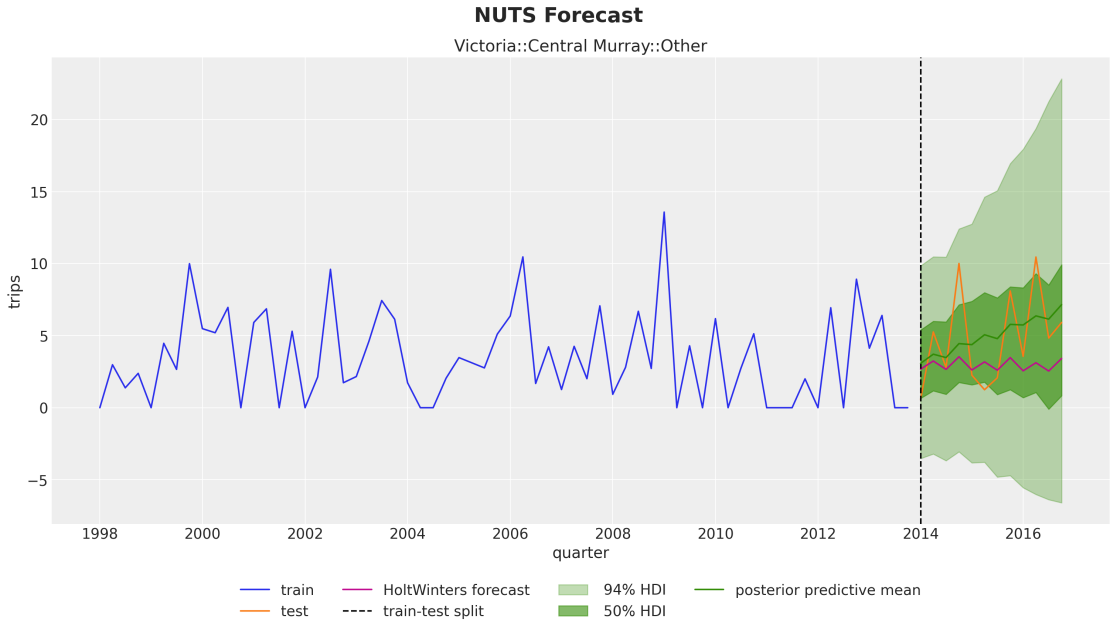


FIGURE 8. Posterior distributions of smoothing parameters across regions. The hierarchical structure shrinks extreme estimates toward the population mean while preserving meaningful heterogeneity.

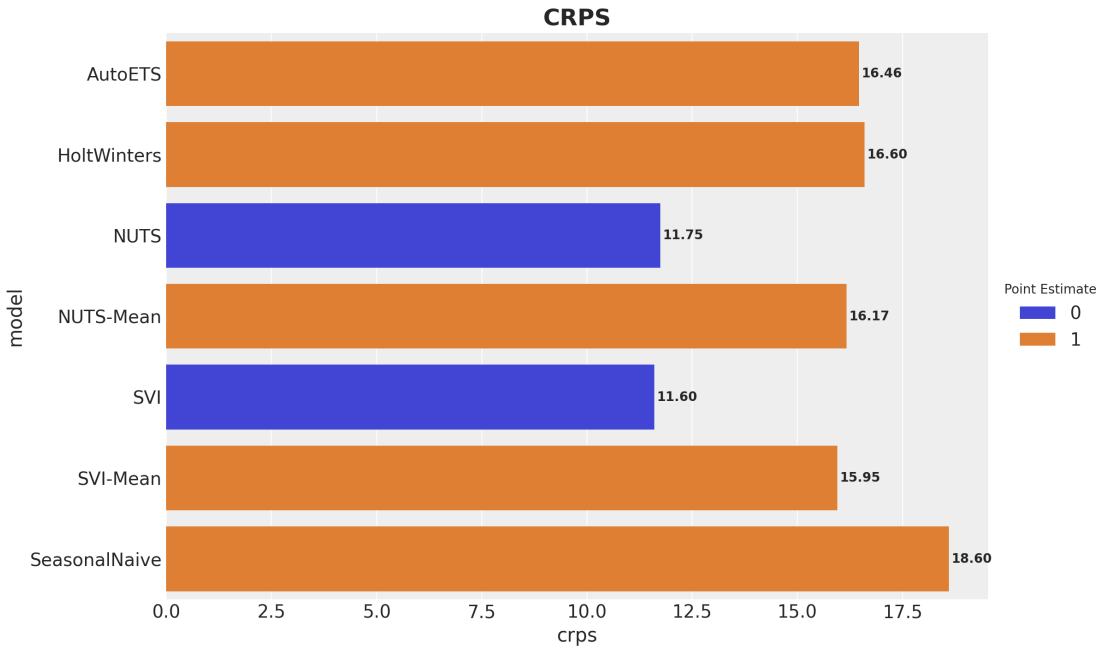


FIGURE 9. Hierarchical forecasts for selected regions with 94% HDIs. The model appropriately quantifies uncertainty, with wider intervals for regions with higher volatility.

6.2. **Baseline Production Model.** For large-scale applications, we developed a baseline model combining local level dynamics with Fourier seasonality and external covariates:

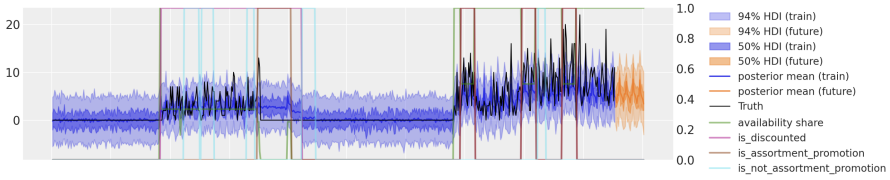


FIGURE 10. Production baseline model architecture: local level for trend, Fourier modes for seasonality, covariates for promotions/discounts, and a global availability factor. This model scales to approximately 40,000 time series in under 10 minutes on GPU.

## 7. INTERMITTENT DEMAND FORECASTING

Intermittent demand—characterized by sporadic, often zero observations—requires specialized forecasting approaches. Standard methods that assume continuous demand systematically fail on such data.

7.1. **Croston's Method.** Croston's method [5] separates the demand size $z_t$ and inter-arrival period $p_t$, forecasting each separately using simple exponential smoothing:

$$\hat{y}_{t+h} = \frac{\hat{z}_{t+h}}{\hat{p}_{t+h}}$$

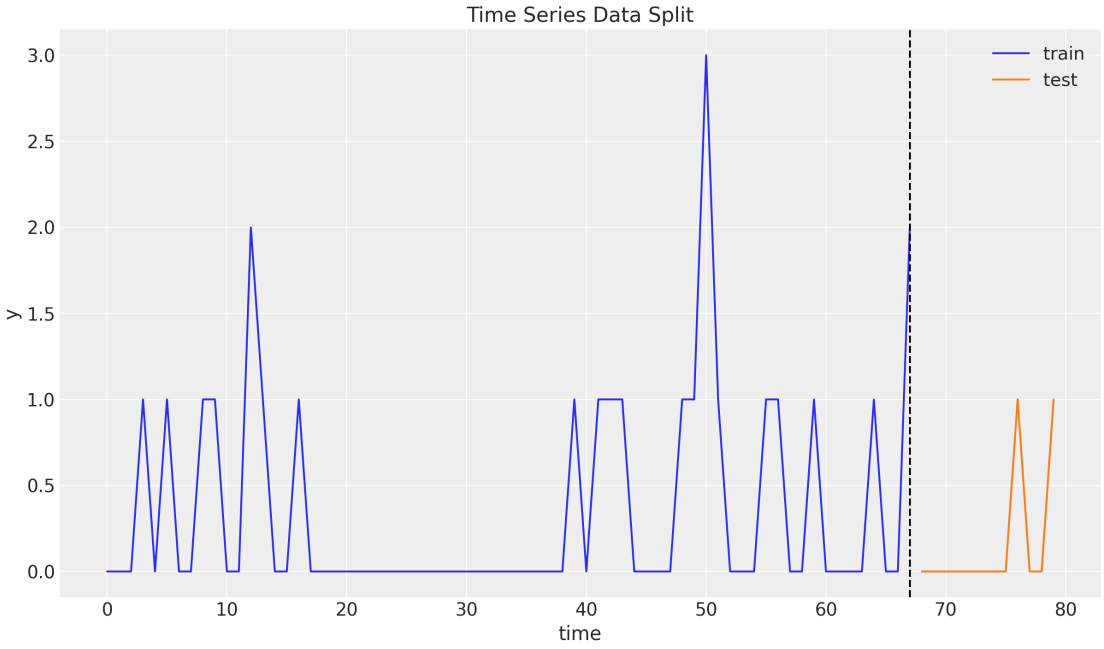


FIGURE 11. Example intermittent demand series showing sporadic non-zero values. The long stretches of zeros interspersed with variable demand sizes motivate the separation approach.

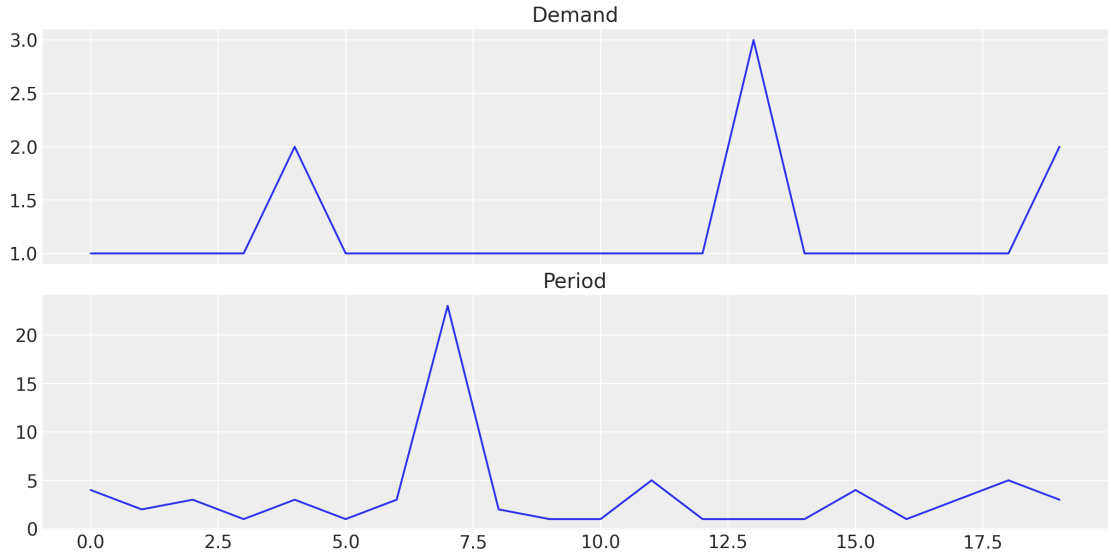


FIGURE 12. Decomposition into demand size (top) and inter-arrival time (bottom) series. Each component can now be modeled with standard exponential smoothing.

The NumPyro implementation uses `scope` to combine two exponential smoothing models:

LISTING 5. Croston's method using scoped models

```
def croston_model(z: ArrayLike, p_inv: ArrayLike, future: int = 0) -> None:
    z_forecast = scope(level_model, "demand")(z, future)
    p_inv_forecast = scope(level_model, "period_inv")(p_inv, future)

    if future > 0:
        numpyro.deterministic("z_forecast", z_forecast)
        numpyro.deterministic("p_inv_forecast", p_inv_forecast)
        numpyro.deterministic("forecast", z_forecast * p_inv_forecast)
```

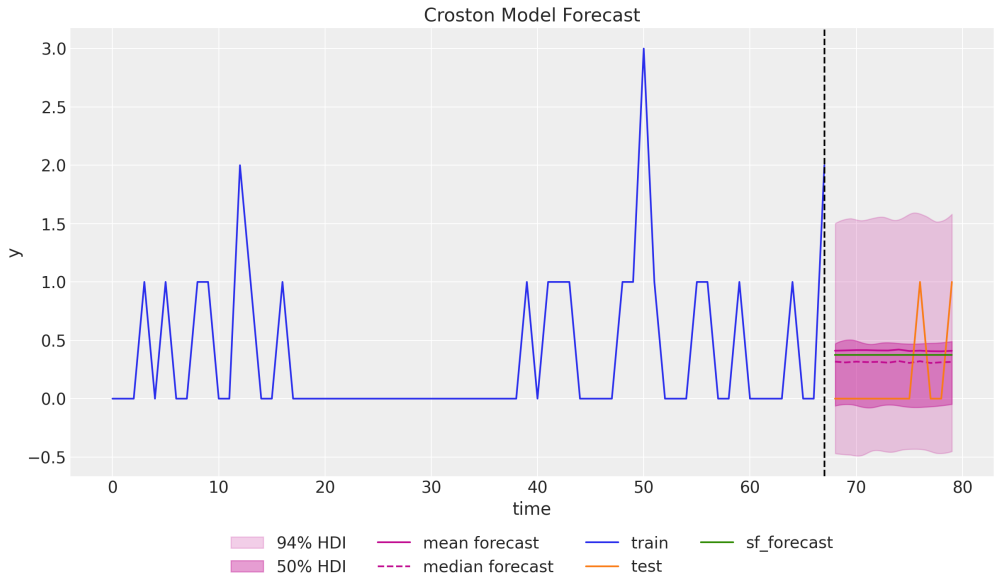


FIGURE 13. Croston's method forecast with uncertainty quantification. The probabilistic formulation provides credible intervals that appropriately reflect the high uncertainty inherent in intermittent demand forecasting.

7.2. **TSB Method.** The Teunter-Syntetos-Babai (TSB) method [22] replaces the inter-arrival period with a demand probability $p_t$, providing potentially better-calibrated forecasts:

$$\hat{y}_{t+h} = \hat{z}_{t+h} \cdot \hat{p}_{t+h}$$

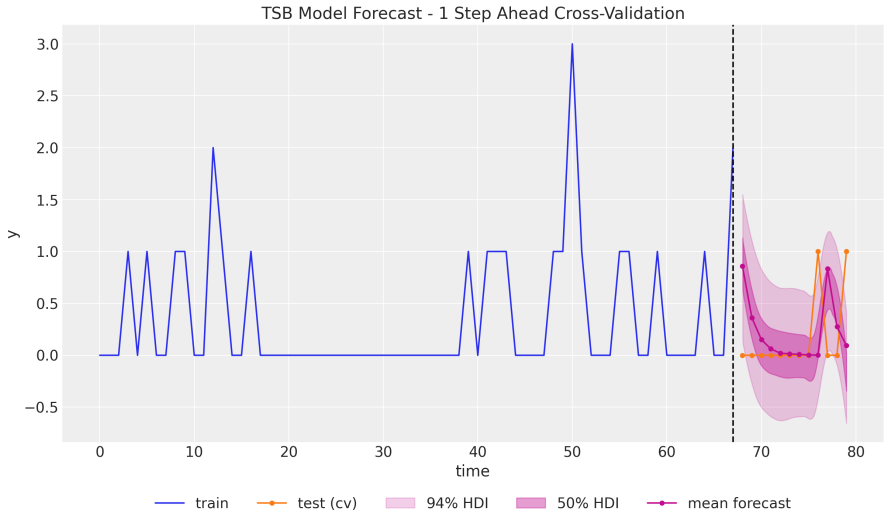


FIGURE 14. TSB method cross-validation results showing one-step-ahead forecasts. The probability formulation often provides more stable forecasts than the period-based Croston approach.

7.3. **Zero-Inflated TSB Model.** For count data with excess zeros, we can modify the TSB model to use a zero-inflated negative binomial distribution:

LISTING 6. Zero-inflated TSB transition function

```
def transition_fn(carry, t):
    z_prev, p_prev = carry
    z_next = ... # Demand size update
    p_next = ... # Probability update

    mu = z_next
    gate = 1 - p_next
    pred = numpyro.sample(
        "pred",
        dist.ZeroInflatedNegativeBinomial2(
            mean=mu, concentration=concentration, gate=gate
        ),
    )
    return (z_next, p_next), pred
```

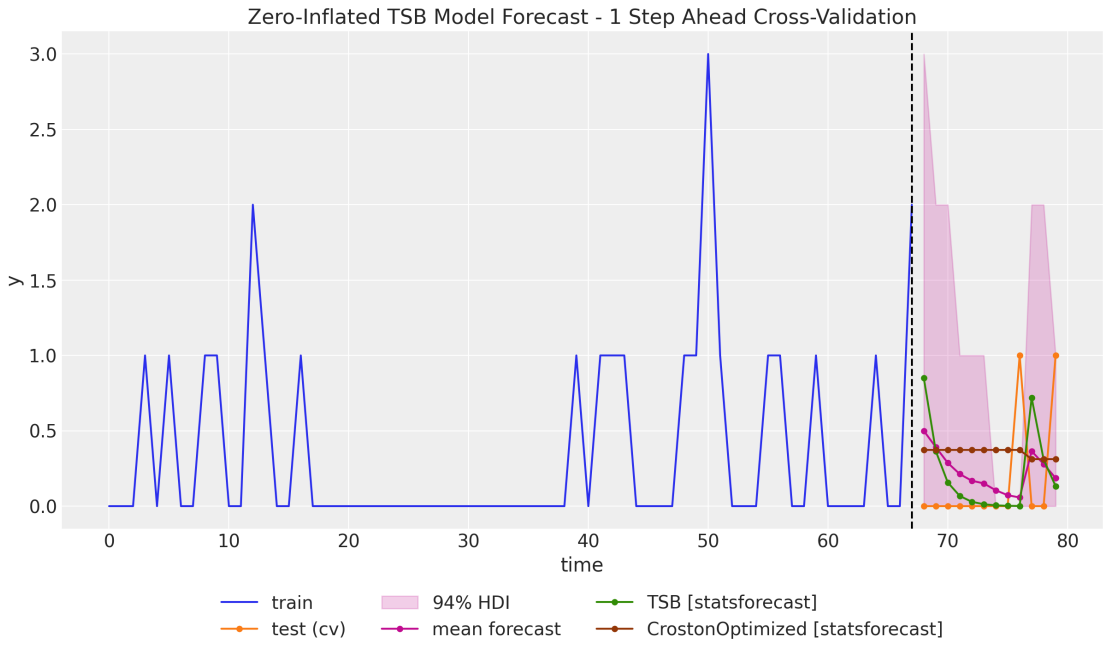


FIGURE 15. Zero-inflated TSB model cross-validation. The explicit zero-inflation component better captures the two-stage nature of intermittent demand: first whether demand occurs, then the demand size conditional on occurrence.

## 8. Availability-Constrained TSB: A Custom Probabilistic Model

This section showcases one of the key advantages of probabilistic programming: the ability to encode domain knowledge directly into model structure. The problem of distinguishing between true zero demand and availability-induced zeros motivates a novel modification to the TSB model.

### 8.1. **The Problem: Why Zeros Happen.**

As discussed by Svetunkov [20], zeros in intermittent time series can arise from two fundamentally different causes:

(1) **True zero demand**: No customer wanted the product
(2) **Availability constraint**: Product was out of stock or unavailable

Standard TSB models treat all zeros equally, causing the non-zero probability to decay regardless of the cause. This leads to systematic underforecasting after stockouts.

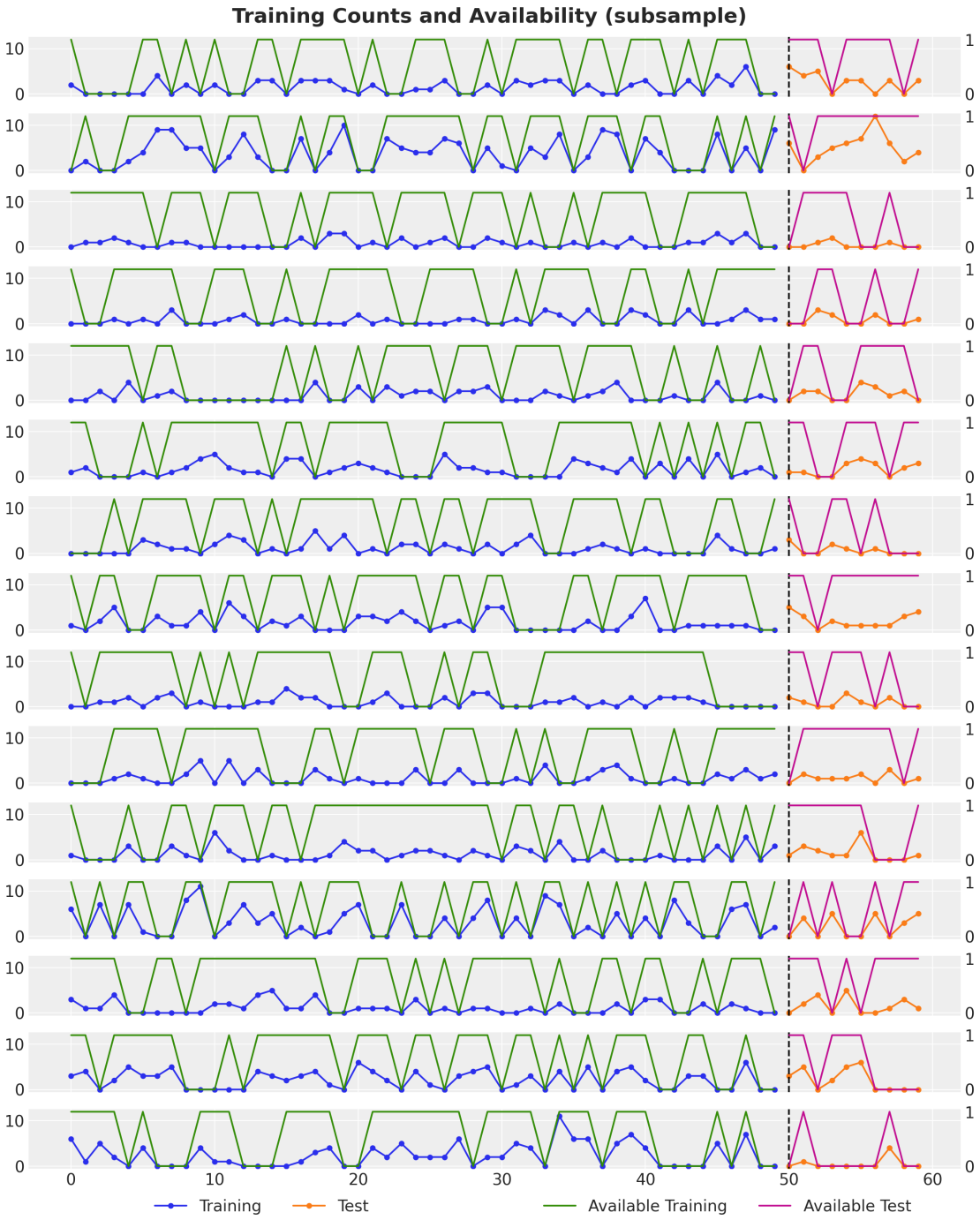


FIGURE 16. Simulated data with availability mask. The red regions indicate periods of unavailability where zeros are not informative about true demand.

8.2. **The Elegant Mathematical Modification.** The key insight is to modify the probability update equation to account for availability:

**Standard TSB** (when $y_t = 0$):

$$z_{t+1} = z_t$$
$$p_{t+1} = (1 - \beta) \cdot p_t$$

**Availability-Constrained TSB** (when $y_t = 0$):

$$z_{t+1} = z_t$$
$$p_{t+1} = (1 - a_t \cdot \beta) \cdot p_t$$

where $a_t \in \{0, 1\}$ indicates availability. When $a_t = 0$ (unavailable), the probability does not decay. Additionally, the forecast incorporates availability:

$$\hat{y}_{t+1} = a_t \cdot z_t \cdot p_t$$

Listing 7. Availability-constrained TSB transition function

```python
def transition_fn(carry, t):
    z_prev, p_prev = carry

    z_next = jnp.where(
        counts[t] > 0,
        z_smoothing * counts[t] + (1 - z_smoothing) * z_prev,
        z_prev,
    )

    p_next = jnp.where(
        counts[t] > 0,
        p_smoothing + (1 - p_smoothing) * p_prev,
        (1 - available[t] * p_smoothing) * p_prev, # Key modification
    )

    mu = z_next * p_next
    pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))
    pred = numpyro.deterministic("pred_obs", available[t] * pred)

    return (z_next, p_next), pred
```

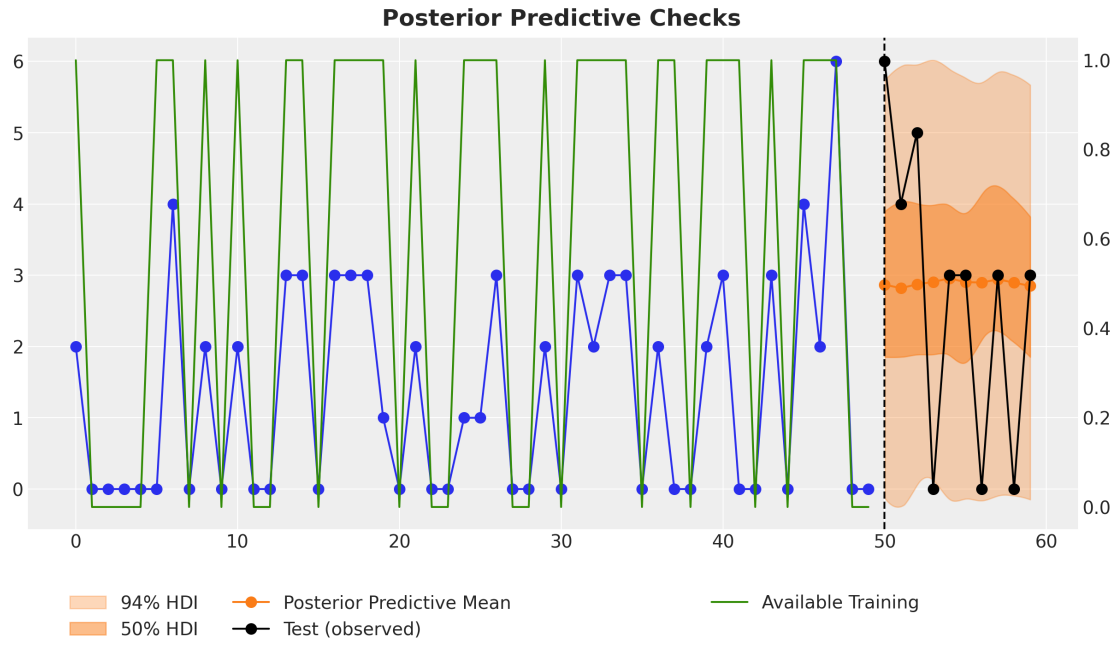


FIGURE 17. Key result: forecast comparison after zeros with availability=0. The standard TSB (left) incorrectly decays the demand probability, while the availability-constrained model (right) maintains the probability, correctly anticipating demand recovery.

8.3. **Why This Matters.** This example illustrates several key advantages of probabilistic programming:

(1) **Domain knowledge integration**: The modification directly encodes business logic into the model structure.
(2) **Minimal change, maximum impact**: A single-line change in the transition function fundamentally alters model behavior.
(3) **Scenario planning enabled**: Setting future availability = 1 forecasts unconstrained demand.
(4) **Preserves model structure**: No new parameters, same inference procedure.
(5) **Scalable**: 1,000 series × 60 time steps in approximately 10 seconds with SVI.

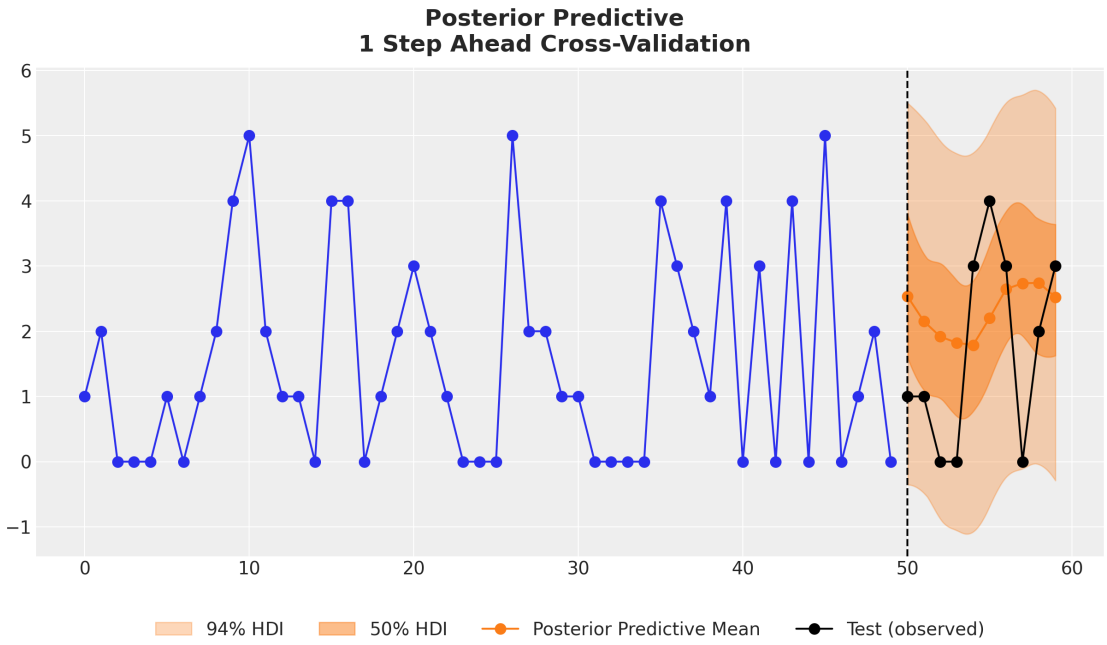


FIGURE 18. Cross-validation results demonstrating the model's ability to adapt forecasts based on availability information across multiple series.

## 9. Censored Demand Forecasting

Stockouts create a different challenge: observed sales are censored—we know demand was *at least* the observed quantity, but true demand may have been higher. Standard models that ignore censoring underestimate true demand.

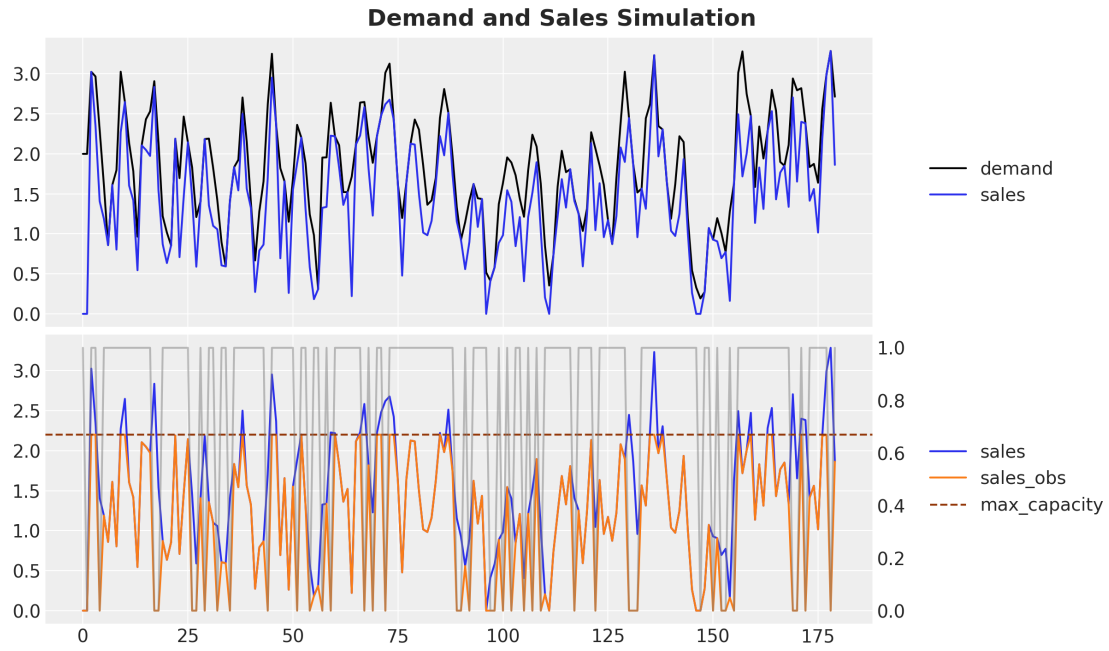


FIGURE 19. Demand data with stockout periods (shaded). During stockouts, observed sales represent a lower bound on true demand, not demand itself.

9.1. **Censored Likelihood.** The censored normal likelihood accounts for observations that are right-censored:

LISTING 8. Censored normal likelihood implementation

```
def censored_normal(loc, scale, y, censored):
    distribution = dist.Normal(loc=loc, scale=scale)
    ccdf = 1 - distribution.cdf(y)
    numpyro.sample(
        "censored_label",
        dist.Bernoulli(probs=ccdf).mask(censored == 1),
        obs=censored
    )
    return numpyro.sample("pred", distribution.mask(censored != 1))
```

For uncensored observations, we use the standard normal likelihood. For censored observations, we contribute only the survival function $P(Y > y)$ to the likelihood.

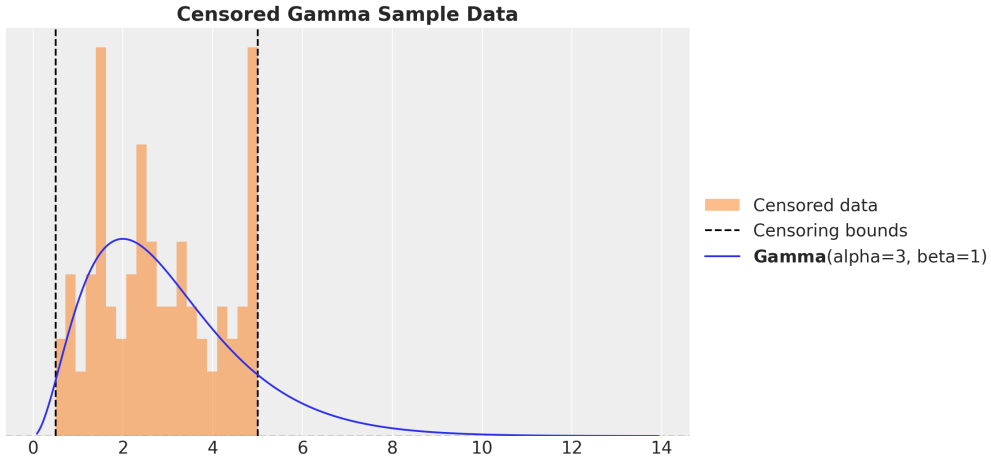


FIGURE 20. Illustration of censoring: when inventory is limited, we observe the inventory level but not the true demand that exceeded it.

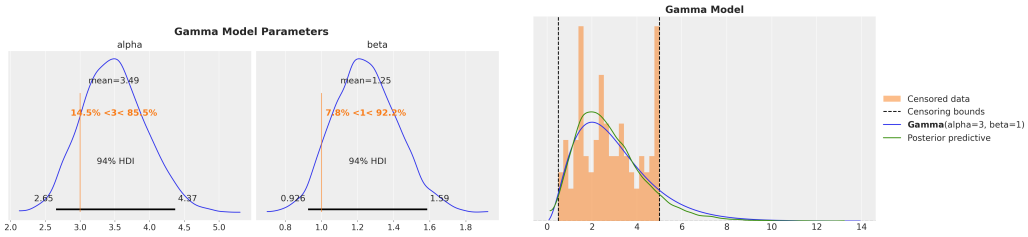


FIGURE 21. Left: Standard model ignoring censoring underestimates parameters. Right: Censored likelihood recovers true parameters, correctly accounting for truncated observations.

9.2. **Censored Time Series Model.** Integrating the censored likelihood into a time series model requires modifying the transition function:

LISTING 9. AR(2) transition with censored likelihood

```
def transition_fn(carry, t):
    y_prev_1, y_prev_2 = carry
    ar_part = phi_1 * y_prev_1 + phi_2 * y_prev_2
    pred_mean = mu + ar_part + seasonal[t]
    # Censored likelihood
    pred = censored_normal(pred_mean, sigma, y[t], censored[t])
    return (pred, y_prev_1), pred
```

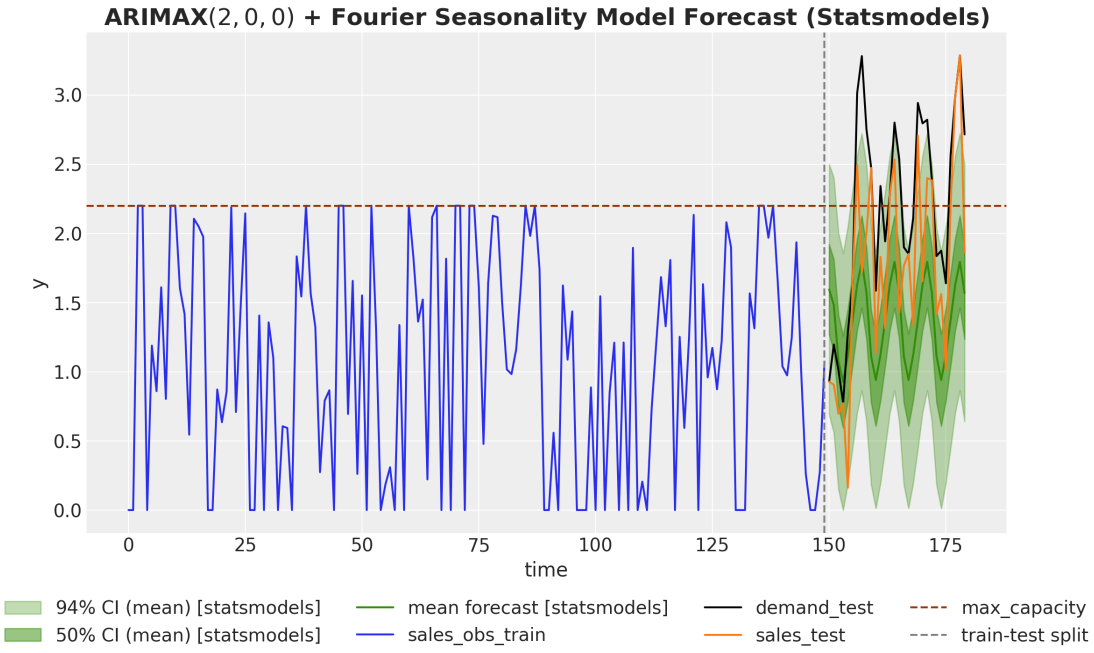


FIGURE 22. Standard ARIMA forecast ignoring censoring. The model underestimates demand during stockout periods and produces biased forecasts.

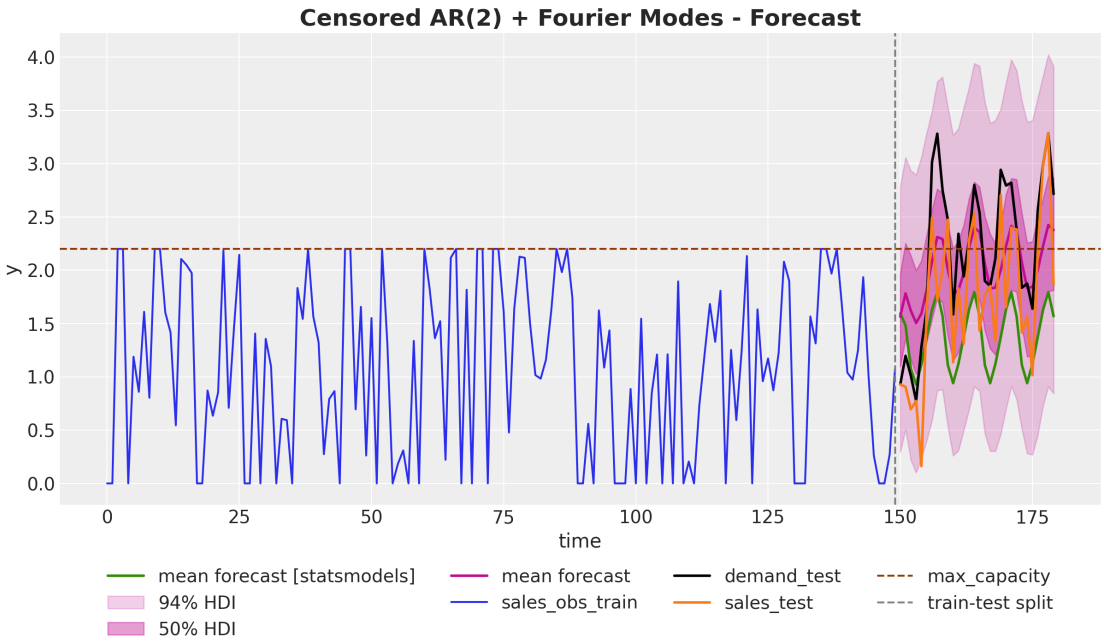


FIGURE 23. Censored ARIMA forecast correctly inferring latent demand. The model produces unbiased forecasts by properly accounting for the censoring mechanism.

## 10. Hierarchical Price Elasticity Models

Price elasticity estimation—measuring how demand responds to price changes—benefits from hierarchical modeling when data is sparse at the individual product level but abundant across products.

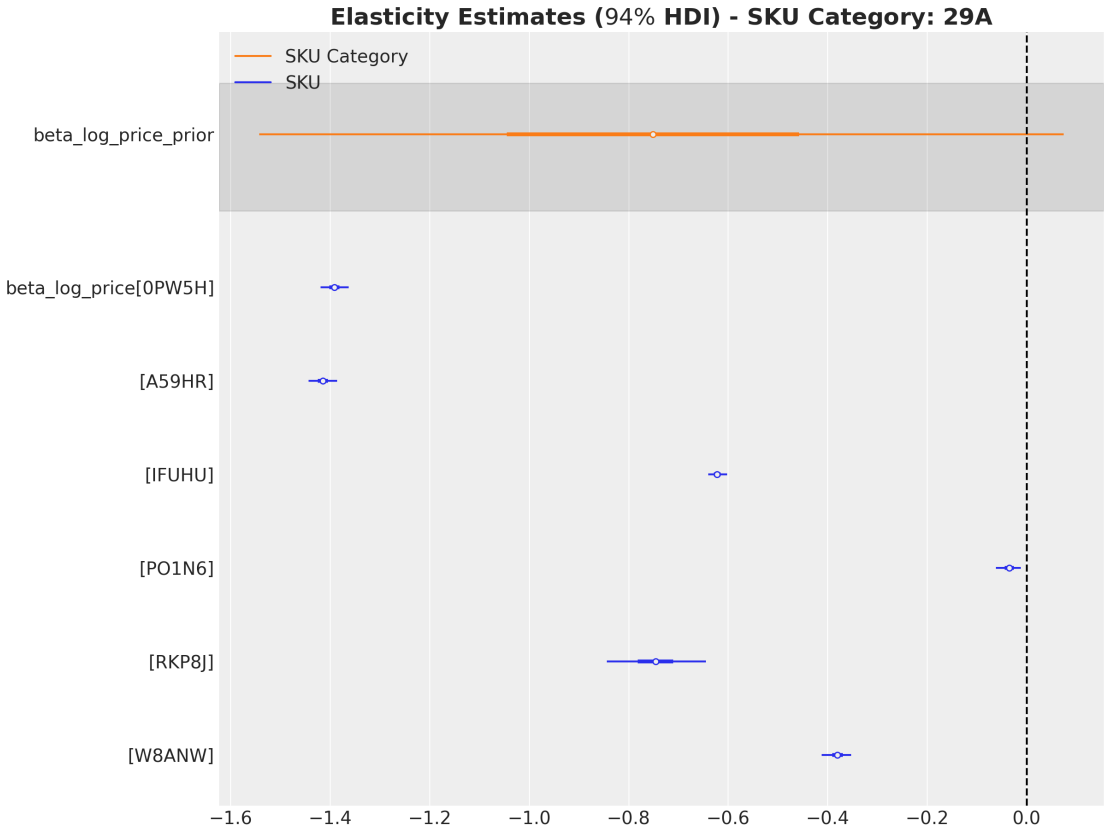


FIGURE 24. Posterior distributions of price elasticities across SKUs. The hierarchical structure shrinks extreme estimates toward the category mean while preserving meaningful product-level heterogeneity.

The hierarchical prior on elasticities takes the form:

$$\begin{aligned} \beta_{\text{price},i} &\sim \text{Normal}(\mu_\beta, \sigma_\beta) \\ \mu_\beta &\sim \text{Normal}(-1, 0.5) \\ \sigma_\beta &\sim \text{HalfNormal}(0.5) \end{aligned}$$

This structure regularizes elasticity estimates for products with limited price variation while allowing products with rich data to deviate from the category mean.

## 11. Model Calibration with Additional Likelihoods

This section showcases a powerful technique for incorporating domain knowledge: treating prior beliefs as additional likelihood terms. The key inspiration comes from the Pyro DLM tutorial [16].

11.1. **The Core Insight: Priors as Likelihoods.** The fundamental technique is elegant: in probabilistic programming, we can inject constraints at specific points by adding observed sample statements. From the Pyro DLM tutorial:

LISTING 10. Calibration pattern from Pyro DLM tutorial

```
# Inject prior terms as if they were likelihoods
for tp, prior in zip(time_points, priors):
    pyro.sample("weight_prior_{}".format(tp),
                dist.Normal(prior, 0.5).to_event(1),
                obs=weight[..., tp:tp+1, 1:])
```

This allows:

(1) Setting informative priors for coefficients at specific time points
(2) Incorporating experimental results (e.g., lift tests)
(3) Constraining model behavior in specific regimes
(4) **Calibrating any latent component, including Gaussian Processes**

11.2. **Key Application: HSGP Calibration for Electricity Demand.** Our primary calibration example demonstrates calibrating a Hilbert Space Gaussian Process (HSGP) latent component for electricity demand forecasting [11].

**Model structure**:

- Linear model for demand with temperature as regressor
- Hour and day-of-week seasonal effects (ZeroSumNormal)
- Matérn 5/2 kernel via HSGP approximation for temperature effect
- Heteroscedastic noise (scale varies with $\sqrt{\text{temperature}}$)
- Student-t likelihood for robustness

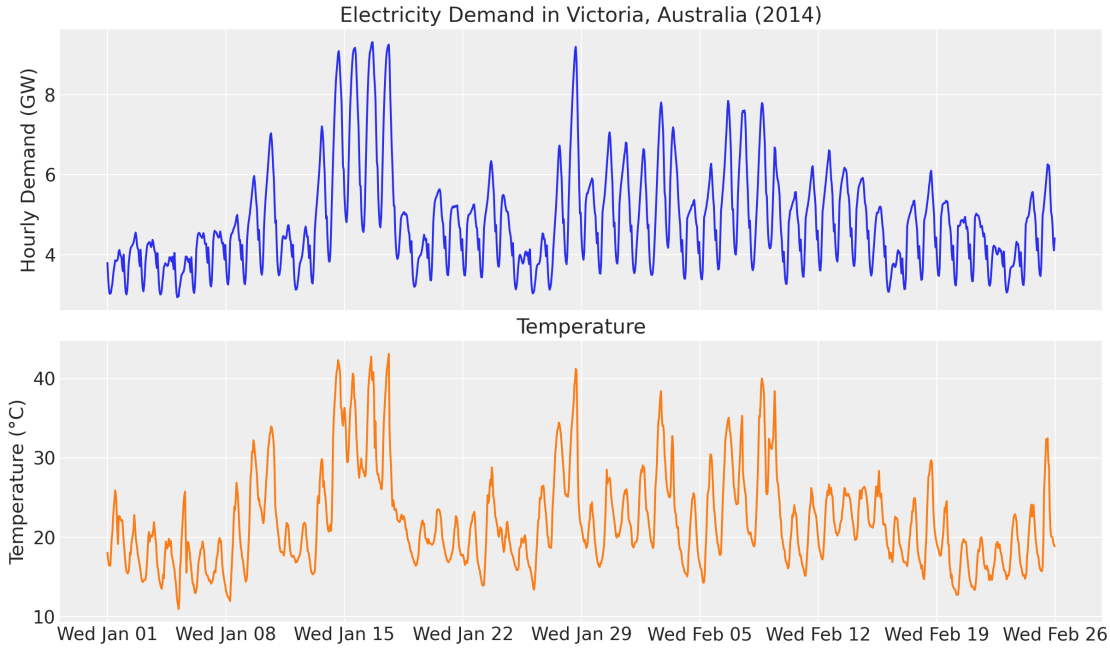


FIGURE 25. Electricity demand data showing the non-linear relationship with temperature. Domain knowledge suggests the effect should stabilize at high temperatures.

**The problem**: At extreme temperatures ($> 32$°C), historical data is sparse, and domain knowledge suggests the temperature effect should stabilize around 0.13.

**The solution**: Calibrate the HSGP latent component:

LISTING 11. HSGP calibration for electricity demand

```
# Temperature effect as HSGP (Matern 5/2 approximation)
beta_temperature = numpyro.deterministic(
    "beta_temperature",
    hsgp_matern(x=temperature, nu=5/2, alpha=alpha,
                length=length_scale, ell=ell, m=m)
)

# Calibrate GP for high temperatures (>32C)
temperature_prior_idx = jnp.where(temperature > 32.0)[0]
if temperature_prior_idx is not None:
    numpyro.sample(
        "temperature_prior",
        dist.Normal(loc=0.13, scale=0.01), # Domain knowledge
        obs=beta_temperature[temperature_prior_idx]
    )
```

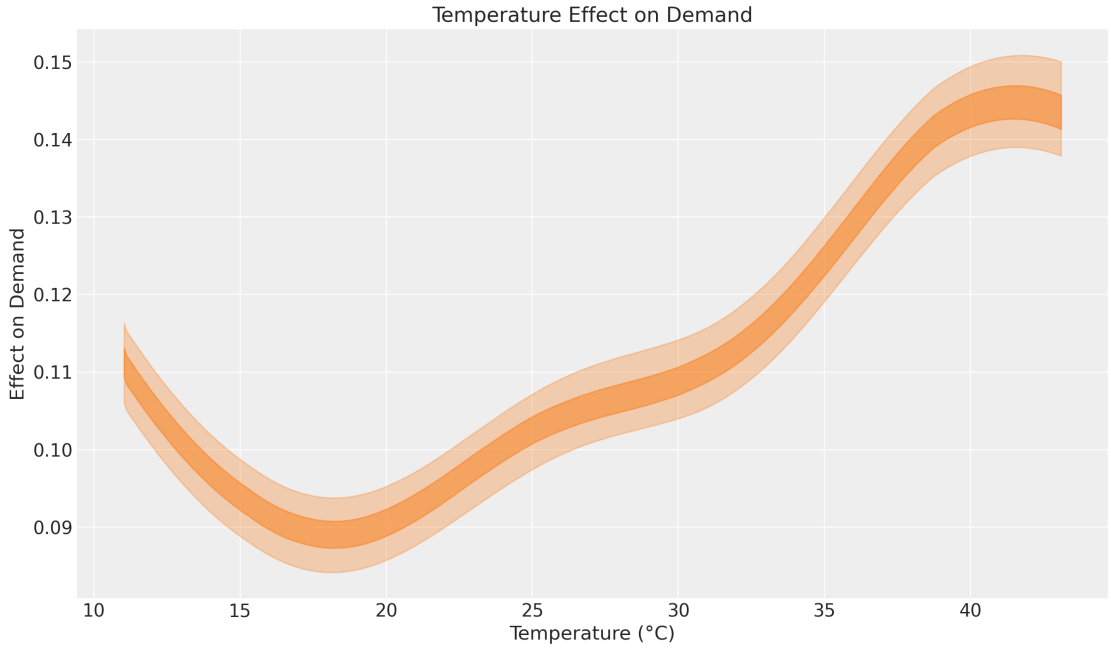


FIGURE 26. Uncalibrated temperature effect from the baseline model. The effect oscillates between 0.11 and 0.15 in the high-temperature regime due to data sparsity.

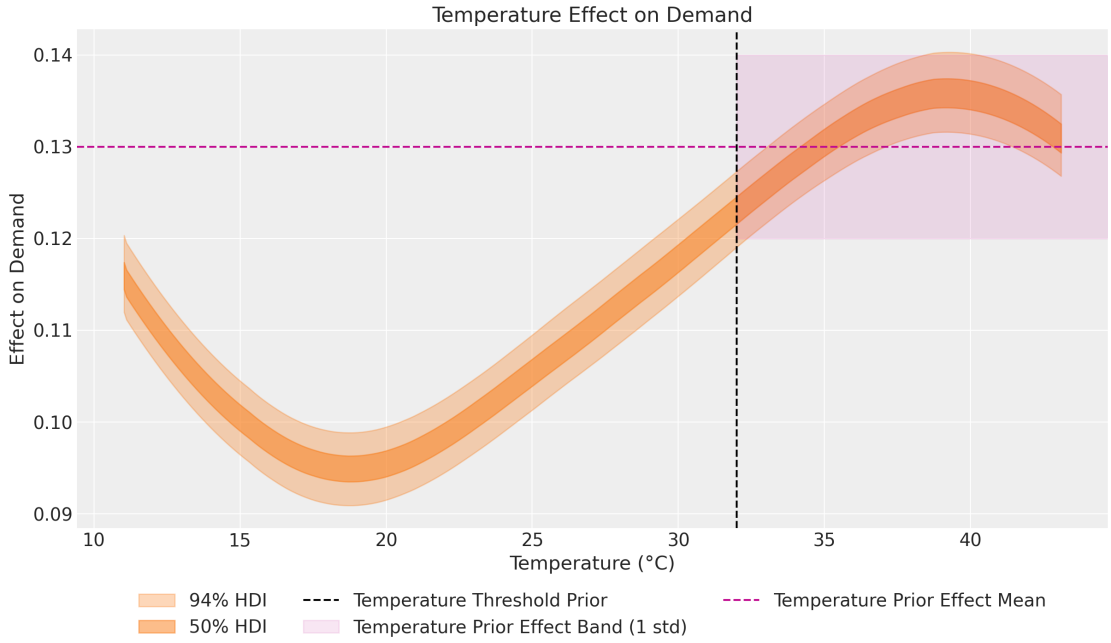


FIGURE 27. Calibrated temperature effect. The domain knowledge constraint stabilizes the effect around 0.13 for temperatures above 32°C while maintaining flexibility elsewhere. Forecast metrics (CRPS) remain essentially unchanged.

11.3. **Why HSGP Calibration is Novel.** Most calibration examples focus on simple linear coefficients. This example demonstrates:

- Calibration works for **any latent component**, including Gaussian Processes
- HSGP provides efficient GP approximation; calibration refines specific regimes
- Combines GP flexibility with domain knowledge interpretability
- Forecast accuracy preserved while improving coefficient estimates

11.4. **Application: MMM Calibration with Lift Tests.** Marketing Mix Models (MMM) suffer from unobserved confounders. Lift tests provide experimental ground truth that can be incorporated as calibration likelihoods [15]:


FIGURE 28. MMM calibration with lift tests. Experimental results constrain media effectiveness estimates, addressing the unobserved confounders problem common in marketing attribution.

## 12. Stochastic Variational Inference

Markov Chain Monte Carlo (MCMC) can be computationally expensive for large-scale forecasting problems involving thousands of time series. Stochastic Variational Inference (SVI) [3] transforms posterior inference into an optimization problem, enabling scalable inference.

12.1. **Core Concepts. Variational family**: Approximate the complex posterior $p(\theta|y)$ with a simpler distribution $q_\phi(\theta)$ parameterized by $\phi$.

**ELBO**: The Evidence Lower Bound serves as the optimization target:

$$\text{ELBO}(\phi) = \mathbb{E}_{q_\phi}[\log p(y|\theta)] - \text{KL}[q_\phi(\theta) \| p(\theta)]$$

The first term encourages the approximation to explain the data well; the second term regularizes toward the prior.

12.2. **NumPyro SVI Workflow:**

LISTING 12. SVI setup with Optax optimizer

```
from numpyro.infer import SVI, Trace_ELBO
from numpyro.infer.autoguide import AutoNormal
import optax

# Define guide (variational family)
guide = AutoNormal(model)

# Define optimizer with Optax
scheduler = optax.linear_onecycle_schedule(
    transition_steps=num_steps,
    peak_value=0.01,
```

```
    pct_start=0.1
)
optimizer = optax.chain(
    optax.clip_by_global_norm(1.0),
    optax.adam(learning_rate=scheduler)
)

# Initialize SVI
svi = SVI(model=model, guide=guide, optim=optimizer, loss=Trace_ELBO())

# Run optimization
svi_result = svi.run(rng_key, num_steps, y=data)
```

12.3. **JAX Ecosystem Integration.** NumPyro integrates seamlessly with the broader JAX ecosystem:

- **Optax**: Composable gradient transformations including Adam, learning rate schedulers, and gradient clipping.
- **Flax NNX**: Neural network modules that can be integrated into probabilistic models (see Section 13).
- **JAX JIT**: Automatic compilation for efficient execution on GPU/TPU.

12.4. **AutoGuides.** NumPyro provides automatic guide construction:

- `AutoNormal`: Mean-field approximation with independent Normal for each parameter
- `AutoMultivariateNormal`: Full covariance approximation
- `AutoLowRankMultivariateNormal`: Low-rank approximation for high-dimensional problems

The automatic handling of constraints and transformations makes SVI accessible without deep expertise in variational inference.

## 13. Hybrid Deep State-Space Models

The final advancement combines neural network components with probabilistic state-space models, enabling learning of complex patterns while maintaining interpretable structure [12].

13.1. **Neural Network Components.** We define embedding and transition networks using Flax NNX:

Listing 13. Neural network components for hybrid model

```
class StationEmbedding(nnx.Module):
    def __init__(self, n_stations: int, embedding_dim: int, rngs: nnx.Rngs):
        self.embedding = nnx.Embed(
            num_embeddings=n_stations,
            features=embedding_dim,
            rngs=rngs
        )
```

```

    def __call__(self, station_idx):
        return self.embedding(station_idx)


class TransitionNetwork(nnx.Module):
    def __init__(self, embedding_dim: int, hidden_dim: int, rngs: nnx.Rngs):
        self.dense1 = nnx.Linear(embedding_dim + 1, hidden_dim, rngs=rngs)
        self.dense2 = nnx.Linear(hidden_dim, 1, rngs=rngs)

    def __call__(self, embedding, level):
        x = jnp.concatenate([embedding, level[..., None]], axis=-1)
        x = nnx.relu(self.dense1(x))
        return self.dense2(x).squeeze(-1)
```

13.2. **Hybrid Model Structure.** The hybrid model augments a local level state-space model with neural network corrections:

Listing 14. Hybrid transition function with NN component

```
def transition_fn(carry, t):
    level_prev = carry

    # Station embedding
    embedding = station_embedding(station_idx)

    # NN correction to drift
    nn_correction = transition_network(embedding, level_prev)

    # Local level update with NN correction
    level = level_prev + drift + nn_correction
    level = jnp.where(
        t < t_max,
        level_smoothing * y[:, t] + (1 - level_smoothing) * level,
        level
    )

    mu = level
    pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))

    return level, pred
```

The neural network learns station-specific adjustments to the level dynamics that cannot be captured by the simple local level model alone.

13.3. **SVI Training.** Training hybrid models requires SVI due to the non-conjugate neural network parameters:

Listing 15. SVI training for hybrid model

```
# Initialize neural network
nn_params = nnx.state(transition_network)

# AutoNormal guide for probabilistic parameters
guide = AutoNormal(hybrid_model, init_loc_fn=init_to_mean)

# Optax optimizer with OneCycle schedule
scheduler = optax.linear_onecycle_schedule(
    transition_steps=20000,
    peak_value=0.005
)
optimizer = optax.adam(learning_rate=scheduler)

svi = SVI(hybrid_model, guide, optimizer, Trace_ELBO())
svi_result = svi.run(rng_key, 20000, y=data, station_idx=stations)
```

13.4. **When Neural Networks Help.** Neural network corrections are most valuable when:

- Multiple related time series share complex non-linear patterns
- Standard state-space models leave systematic residual structure
- Sufficient data exists to learn embeddings without overfitting

For simpler problems, pure state-space models often suffice and provide better interpretability.

## 14. Conclusion

This article has presented a comprehensive guide to probabilistic time series forecasting with NumPyro, progressing from foundational concepts to advanced hybrid architectures.

14.1. **Key Contributions.** We demonstrated several powerful techniques:

(1) **The scan pattern**: A universal foundation for state-space models in NumPyro, enabling efficient implementation of recursive relationships.
(2) **Hierarchical modeling**: Sharing information across related time series while respecting individual heterogeneity, particularly valuable for sparse data.
(3) **Custom likelihoods**: Censored and zero-inflated distributions that accurately model real-world data generating processes.
(4) **Domain knowledge integration**: Two showcase applications—availability-constrained TSB and HSGP calibration—demonstrate how probabilistic programming enables encoding business logic directly into model structure.
(5) **Scalable inference**: SVI with Optax integration enables training on thousands of time series efficiently.
(6) **Neural network integration**: Hybrid models combine interpretable state-space structure with flexible neural network components.

14.2. **Practical Recommendations.** Based on our experience applying these methods in production:

- Start simple: exponential smoothing models often provide strong baselines.
- Use hierarchical structure when data is sparse for individual series.
- Encode domain knowledge explicitly through model structure or calibration.
- Validate uncertainty calibration, not just point forecast accuracy.
- Use SVI for development iteration, MCMC for final inference when feasible.

14.3. **Acknowledgments.** We gratefully acknowledge the NumPyro core developers—Du Phan, Neeraj Pradhan, and Martin Jankowiak—for creating and maintaining this excellent library. The Pyro team at Uber AI Labs, including Eli Bingham, Fritz Obermeyer, and Noah Goodman, laid the foundation with Pyro [2]. The JAX team at Google enables the performance that makes these methods practical. Finally, the open-source community's contributions through issues, discussions, and pull requests continuously improve these tools.

## References

[1] Alexandrov, A., Benidis, K., Bohlke-Schneider, M., et al. GluonTS: Probabilistic and neural time series modeling in Python. *Journal of Machine Learning Research 21*, 116 (2020), 1–6.

[2] Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P. A., Horsfall, P., and Goodman, N. D. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res. 20* (2019), 28:1–28:6.

[3] Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. Variational inference: A review for statisticians. *Journal of the American Statistical Association 112*, 518 (2017), 859–877.

[4] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018.

[5] Croston, J. D. Forecasting and stock control for intermittent demands. *Journal of the Operational Research Society 23*, 3 (1972), 289–303.

[6] Durbin, J., and Koopman, S. J. *Time Series Analysis by State Space Methods*, 2nd ed. Oxford University Press, 2012.

[7] Harvey, A. C. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1989.

[8] Hyndman, R. J., Koehler, A. B., Ord, J. K., and Snyder, R. D. *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Science & Business Media, 2008.

[9] Hyndman, R. J., Koehler, A. B., Snyder, R. D., and Grose, S. A state space framework for automatic forecasting using exponential smoothing methods. *International Journal of Forecasting 18*, 3 (2002), 439–454.

[10] Lim, B., Arik, S. Ö., Loeff, N., and Pfister, T. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting 37*, 4 (2021), 1748–1764.

[11] Orduz, J. Electricity Demand Forecast: Dynamic Time-Series Model with Prior Calibration. `https://juanitorduz.github.io/electricity_forecast_with_priors/`, 2024.

[12] Orduz, J. From Pyro to NumPyro: Forecasting Hierarchical Models - Part III. `https://juanitorduz.github.io/numpyro_hierarchical_forecasting_3/`, 2024.

[13] Oreshkin, B. N., Carpov, D., Chapados, N., and Bengio, Y. N-BEATS: Neural basis expansion analysis for interpretable time series forecasting. In *International Conference on Learning Representations* (2020).

[14] Phan, D., Pradhan, N., and Jankowiak, M. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

[15] PyMC-Labs Development Team. PyMC-Marketing: Lift Test Calibration. https://www.pymc-marketing.io/en/stable/notebooks/mmm/mmm_lift_test.html, 2024.
[16] Pyro Development Team. Forecasting with Dynamic Linear Model (DLM). https://pyro.ai/examples/forecasting_dlm.html, 2020.
[17] Salinas, D., Flunkert, V., Gasthaus, J., and Januschowski, T. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting 36*, 3 (2020), 1181–1191.
[18] Smyl, S., Bergmeir, C., Ranganathan, S., et al. Local and global trend Bayesian exponential smoothing models. *International Journal of Forecasting* (2024).
[19] Svetunkov, I. *Forecasting and Analytics with the Augmented Dynamic Adaptive Model (ADAM).* Chapman and Hall/CRC, 2023.
[20] Svetunkov, I. Why Zeroes Happen? https://openforecast.org/2024/11/18/why-zeroes-happen/, 2024.
[21] Taylor, S. J., and Letham, B. Forecasting at scale. *The American Statistician 72*, 1 (2018), 37–45.
[22] Teunter, R. H., Syntetos, A. A., and Babai, M. Z. Intermittent demand: Linking forecasting to inventory obsolescence. *European Journal of Operational Research 214*, 3 (2011), 606–615.

*Email address*: juanitorduz@gmail.com
*URL*: https://juanitorduz.github.io/

Berlin, Germany