

Demand forecasting: from business question to delivery evidence

Interview case study | privacy-safe synthetic data

This notebook demonstrates how I would take a demand-planning use case from an explainable baseline to a validated model, while keeping the delivery path simple enough for a lean AI team to operate.

Executive takeaway: the model is not the product. The product is a repeatable decision process: define the question, establish a trusted baseline, validate honestly, and create a clear path from manual operation to future orchestration.

1. The delivery question

A planning team needs a 12-month demand outlook for each product and region. The public version uses synthetic data only; the architecture mirrors the kind of delivery pattern a player/coach AI lead must establish:

- **Business value:** support planning with a consistent forward view.
- **Trust mechanism:** start with a baseline stakeholders can calculate themselves.
- **Technical decision:** add a model only if time-based validation earns the complexity.
- **Operating model:** begin with a manual trigger, then expose the same boundary to orchestration when the process is stable.

This is intentionally a small, transparent slice of a production delivery system—not an AutoML catalogue or a cloud-specific demo.

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

SEED = 7
HORIZON = 12
plt.style.use('seaborn-v0_8-whitegrid')
```

```

COLORS = {'actual': '#16324F', 'baseline': '#F28E2B', 'model': '#2A9D8F', 'a
rng = np.random.default_rng(SEED)
dates = pd.date_range('2020-01-31', periods=60, freq='ME')
rows = []
for sku in ['sku_a', 'sku_b', 'sku_c']:
    for region in ['north', 'south']:
        base = rng.uniform(80, 160)
        for i, date in enumerate(dates):
            seasonal = 1 + 0.18 * np.sin(2 * np.pi * date.month / 12)
            trend = 1 + 0.003 * i
            shock = 1.25 if date.month == 11 else 1.0
            demand = max(0, base * seasonal * trend * shock + rng.normal(0,
            rows.append((date, sku, region, demand))
df = pd.DataFrame(rows, columns=['date', 'sku', 'region', 'demand'])
series = df.query("sku == 'sku_a' and region == 'north'").reset_index(drop=T
print(f'{len(df):,} synthetic observations | {df.sku.nunique()} SKUs | {df.r

```

360 synthetic observations | 3 SKUs | 2 regions | 60 months per series

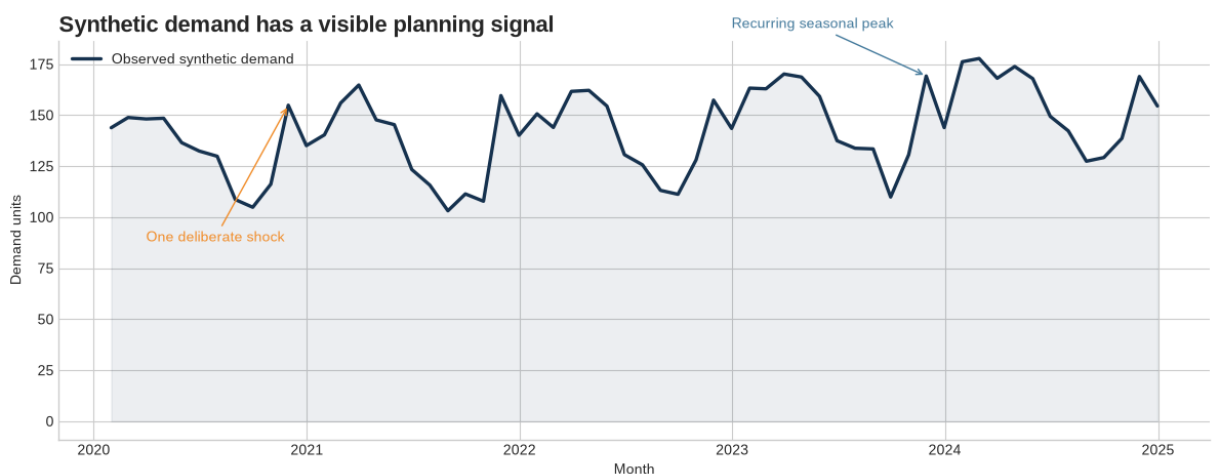
Reading the synthetic signal

The data deliberately contains three interpretable ingredients: gradual trend, recurring seasonality, and a November demand shock. That makes the experiment useful for explaining model behavior without pretending synthetic data proves a real business outcome.

```

In [2]: fig, ax = plt.subplots(figsize=(12, 4.8))
ax.plot(series.date, series.demand, color=COLORS['actual'], linewidth=2.4, l
ax.fill_between(series.date, series.demand, color=COLORS['actual'], alpha=0.
ax.set_title('Synthetic demand has a visible planning signal', loc='left', f
ax.set_ylabel('Demand units')
ax.set_xlabel('Month')
ax.legend(frameon=False, loc='upper left')
ax.annotate('Recurring seasonal peak', xy=(series.date.iloc[46], series.dema
ax.annotate('One deliberate shock', xy=(series.date.iloc[10], series.demand.
ax.spines[['top', 'right']].set_visible(False)
plt.tight_layout()
plt.show()

```



2. Baseline first: a forecast stakeholders can audit

The baseline uses the previous eight observed months, removes one minimum and one maximum, and averages the remaining six:

$$\text{baseline} = \frac{\sum(\text{last } 8) - \min(\text{last } 8) - \max(\text{last } 8)}{6}$$

This is intentionally modest. It reduces sensitivity to one-off spikes while remaining easy to reproduce in a spreadsheet. It also gives the delivery team a reference point for deciding whether model complexity is justified.

```
In [3]: def trimmed_mean(values):
        values = np.asarray(values, dtype=float)
        if len(values) != 8:
            raise ValueError('baseline requires exactly 8 observations')
        return (values.sum() - values.min() - values.max()) / 6

        baseline_value = trimmed_mean(series.demand.tail(8))
        baseline_preview = pd.DataFrame({'month': series.date.tail(8), 'observed_dem
        baseline_preview['included_in_average'] = ~baseline_preview.observed_demand.
        display(baseline_preview.style.format({'observed_demand': '{:.1f}')).set_cap
        print(f'Baseline level for the next horizon: {baseline_value:.1f} demand uni
```

The last eight observations used by the baseline

	month	observed_demand	included_in_average
52	2024-05-31 00:00:00	167.9	True
53	2024-06-30 00:00:00	149.4	True
54	2024-07-31 00:00:00	142.3	True
55	2024-08-31 00:00:00	127.4	False
56	2024-09-30 00:00:00	129.2	True
57	2024-10-31 00:00:00	138.6	True
58	2024-11-30 00:00:00	168.9	False
59	2024-12-31 00:00:00	154.4	True

Baseline level for the next horizon: 146.9 demand units

Baseline observation

The baseline is not expected to capture every seasonal movement. Its job is to be stable, explainable, and difficult to game. If a more

complex model cannot beat it consistently in time-based validation, the right delivery decision is to keep the baseline.

```
In [4]: def smape(actual, predicted):
    actual = np.asarray(actual, dtype=float)
    predicted = np.asarray(predicted, dtype=float)
    denominator = np.abs(actual) + np.abs(predicted)
    return np.mean(np.divide(2 * np.abs(actual - predicted), denominator, ou

def feature_frame(frame):
    return frame.assign(
        month=frame.date.dt.month,
        lag_1=frame.demand.shift(1),
        lag_3=frame.demand.shift(3),
        lag_12=frame.demand.shift(12),
    ).dropna()

def recursive_forecast(model, history, horizon):
    history = [{'date': row.date, 'demand': row.demand} for row in history.i
    predictions = []
    for step in range(horizon):
        next_date = history[-1]['date'] + pd.offsets.MonthEnd(1)
        features = pd.DataFrame([
            'month': next_date.month,
            'lag_1': history[-1]['demand'],
            'lag_3': history[-3]['demand'],
            'lag_12': history[-12]['demand'],
        ])
        prediction = float(model.predict(features)[0])
        predictions.append((next_date, prediction))
        history.append({'date': next_date, 'demand': prediction})
    return pd.DataFrame(predictions, columns=['date', 'demand'])

cutoffs = range(36, len(series) - HORIZON + 1, 6)
results = []
for cutoff in cutoffs:
    train = series.iloc[:cutoff].copy()
    test = series.iloc[cutoff:cutoff + HORIZON].copy()
    baseline = np.repeat(trimmed_mean(train.demand.tail(8)), len(test))
    train_features = feature_frame(train)
    model = HistGradientBoostingRegressor(random_state=SEED, max_iter=150)
    model.fit(train_features[['month', 'lag_1', 'lag_3', 'lag_12']], train_f
    model_forecast = recursive_forecast(model, train, len(test))
    results.append({
        'cutoff': train.date.iloc[-1],
        'baseline_smape': smape(test.demand, baseline),
        'model_smape': smape(test.demand, model_forecast.demand),
        'baseline_mae': mean_absolute_error(test.demand, baseline),
        'model_mae': mean_absolute_error(test.demand, model_forecast.demand)
    })
results = pd.DataFrame(results)
results['winner'] = np.where(results.model_smape < results.baseline_smape, '
display(results.style.format({'baseline_smape': '{:.1%}', 'model_smape': '{:
```

Rolling-origin validation: every row predicts a future 12-month window

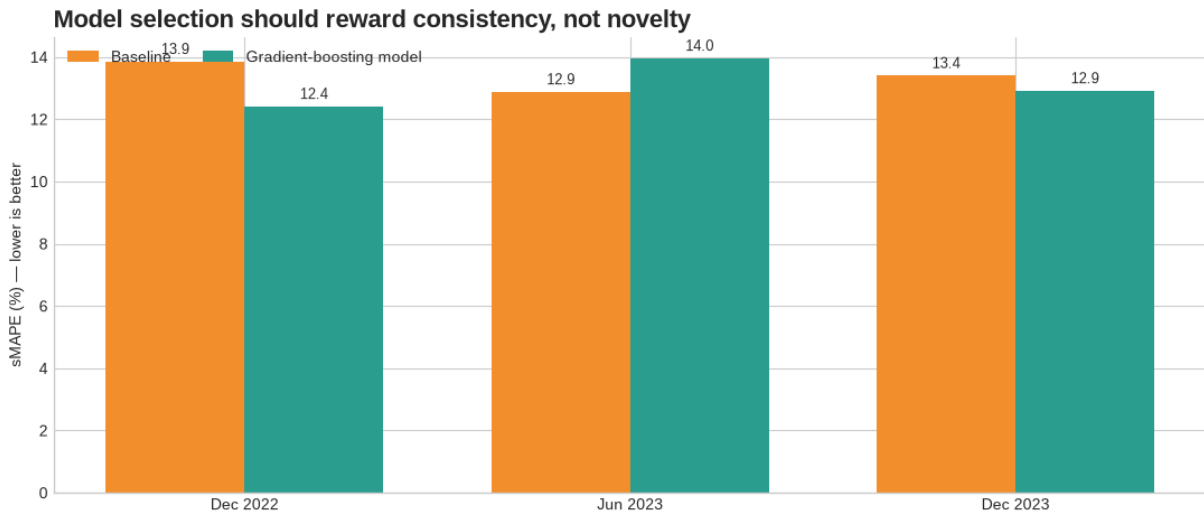
	cutoff	baseline_smape	model_smape	baseline_mae	model_mae	winner
0	2022-12-31 00:00:00	13.9%	12.4%	20.1	18.1	Model
1	2023-06-30 00:00:00	12.9%	14.0%	19.3	20.9	Baseline
2	2023-12-31 00:00:00	13.4%	12.9%	20.4	19.7	Model

3. Validation is the governance mechanism

A random train/test split would let future information leak into the past. Instead, each validation window trains only on data available before its cutoff and predicts the next 12 months. The selection rule is simple: choose the lower sMAPE, but inspect consistency rather than celebrating one favorable window.

That matters operationally. A delivery lead needs evidence that can survive questions from business stakeholders, security/risk partners, and executives—not just a model score from a convenient split.

```
In [5]: x = np.arange(len(results))
width = 0.36
fig, ax = plt.subplots(figsize=(11, 4.8))
ax.bar(x - width / 2, results.baseline_smape * 100, width, color=COLORS['baseline'])
ax.bar(x + width / 2, results.model_smape * 100, width, color=COLORS['model'])
ax.set_xticks(x, results.cutoff.dt.strftime('%b %Y'))
ax.set_ylabel('sMAPE (%) - lower is better')
ax.set_title('Model selection should reward consistency, not novelty', loc='right')
ax.legend(frameon=False, ncol=2, loc='upper left')
ax.spines[['top', 'right']].set_visible(False)
for bars in ax.containers:
    ax.bar_label(bars, fmt='%.1f', padding=3, fontsize=9)
plt.tight_layout()
plt.show()
print(f"Model wins {sum(results.winner == 'Model')} of {len(results)} validation windows")
```



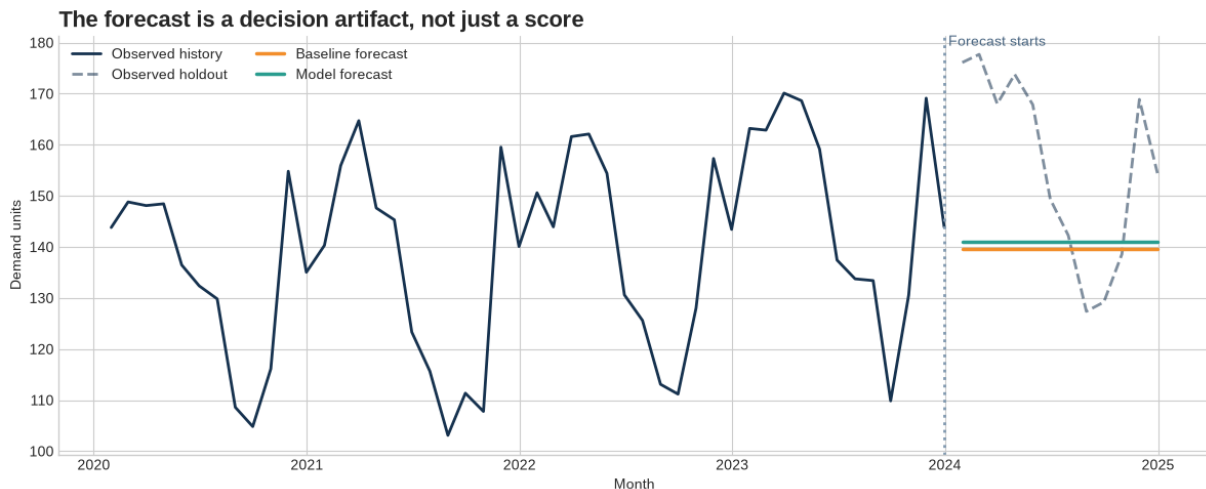
Model wins 2 of 3 validation windows; the baseline wins 1.

Validation observation

The model does not win every window. That is useful evidence, not a failure of the notebook: it shows why a production decision needs a baseline, rolling validation, and an explicit acceptance rule. The next step in a real use case would be to align the acceptance threshold with planning cost, service risk, and stakeholder tolerance—not to add models automatically.

```
In [6]: train = series.iloc[:-HORIZON].copy()
actual = series.iloc[-HORIZON:].copy()
train_features = feature_frame(train)
final_model = HistGradientBoostingRegressor(random_state=SEED, max_iter=150)
final_model.fit(train_features[['month', 'lag_1', 'lag_3', 'lag_12']], train)
model_forecast = recursive_forecast(final_model, train, HORIZON)
baseline_forecast = pd.DataFrame({'date': actual.date, 'demand': trimmed_mean(
    train.demand.rolling(12).median(), 0.1)

fig, ax = plt.subplots(figsize=(12, 5))
ax.plot(train.date, train.demand, color=COLORS['actual'], linewidth=2, label='actual')
ax.plot(actual.date, actual.demand, color=COLORS['actual'], linewidth=2, label='actual')
ax.plot(baseline_forecast.date, baseline_forecast.demand, color=COLORS['baseline'], linewidth=2, label='baseline')
ax.plot(model_forecast.date, model_forecast.demand, color=COLORS['model'], linewidth=2, label='model')
ax.axvline(train.date.iloc[-1], color='#829AB1', linestyle=':', linewidth=2)
ax.text(train.date.iloc[-1], ax.get_ylim()[1], 'Forecast starts', color='#4F81BD', fontweight='bold')
ax.set_title('The forecast is a decision artifact, not just a score', loc='right')
ax.set_ylabel('Demand units')
ax.set_xlabel('Month')
ax.legend(frameon=False, ncol=2, loc='upper left')
ax.spines[['top', 'right']].set_visible(False)
plt.tight_layout()
plt.show()
```



5. Build vs. buy: is a foundation forecasting model worth it?

Gradient boosting beat the baseline in 2 of 3 validation windows above – encouraging, but not enough evidence to declare a production winner. Before adding more classical-model complexity, the pragmatic delivery question is different: **would a pretrained time-series foundation model (e.g. Google's TimesFM) do better out of the box, with zero training?**

This matters for a lean AI team because it changes the trade-off from "tune a model" to "evaluate a build vs. buy decision":

- **Zero-shot, no training** – TimesFM forecasts directly from history, no fitting step.
- **Heavier operationally** – a ~500M-parameter model pulled from Hugging Face, needs `torch` and `transformers`, and meaningfully more latency and memory than gradient boosting.
- **Reproducibility cost** – this case study otherwise runs fully offline in seconds; a foundation model breaks that guarantee unless weights are cached ahead of time.

The cell below is **optional and skipped by default** (set `RUN_TIMESFM=1` to execute it) so the tracked notebook stays fast and runnable without model downloads or GPU access. This mirrors how I would gate an expensive candidate in a real pipeline: available for evaluation, not on the critical path by default.

```
In [7]: import os
```

```
RUN_TIMESFM = os.environ.get('RUN_TIMESFM') == '1'
timesfm_smape = None
```

```

if RUN_TIMESFM:
    import torch
    from transformers import TimesFmModelForPrediction

    # google/timesfm-2.0-500m-pytorch: zero-shot, no fine-tuning, monthly fr
    tfm = TimesFmModelForPrediction.from_pretrained(
        'google/timesfm-2.0-500m-pytorch', dtype=torch.float32, device_map='
    )
    past = torch.tensor(train.demand.values, dtype=torch.float32).unsqueeze(
    freq = torch.tensor([2], dtype=torch.long)
    with torch.no_grad():
        out = tfm(past_values=past, freq=freq, return_dict=True)
        timesfm_forecast = out.mean_predictions[0, :HORIZON].cpu().numpy()
        timesfm_smape = smape(actual.demand.to_numpy(), timesfm_forecast)
        print(f'TimesFM (zero-shot) SMAPE on the final holdout: {timesfm_smape:.
    else:
        print('Skipped: set RUN_TIMESFM=1 to download weights and run the founda

```

Skipped: set RUN_TIMESFM=1 to download weights and run the foundation-model comparison.

Build-vs-buy observation

Whether or not TimesFM is run here, the delivery decision is the same shape: compare the candidate's accuracy gain against its operational cost (dependency weight, latency, offline reproducibility, and team familiarity), not just its headline metric. For this use case, I would only adopt a foundation model in production if it beat both the baseline **and** gradient boosting consistently across many rolling windows – enough to justify the added infrastructure and the loss of full offline reproducibility. Until that evidence exists, the lighter pipeline stays the default, and TimesFM stays available on the bench as an evaluated, not adopted, candidate.

4. Delivery implications for a lean AI task force

What would ship first: a documented, manually triggered workflow with a baseline, a validated model candidate, a forecast output, and a small set of checks that make failures visible.

What I would standardize for the team:

1. A reusable problem brief: decision, horizon, grain, owner, and acceptance metric.

2. A baseline-before-model rule so every use case has a credible comparison point.
3. Time-aware validation and an explicit model-selection record.
4. A handoff boundary that can remain manual while adoption is proved, then move to Airflow or another orchestrator.
5. A privacy review before public or cross-entity reuse.

This is the player/coach signal: build enough of the solution to prove the path, while creating a delivery pattern that other engineers can reuse.

Conclusion

The synthetic experiment is deliberately modest. Its value is the operating discipline around the model: explainable baseline, honest backtest, visible trade-off, and a route from experiment to production ownership.

Decision: keep the lowest-sMAPE approach from rolling validation, document the result, and only increase complexity when the business value and operational readiness justify it.

All data and numeric results in this notebook are synthetic and illustrative.