Title

Author: Ivan Bhargava, Kanav Bhardwaj, Nikunj Mahajan
Computer Science, IIIT-Bangalore

Abstract:

Experiment Details:

Synthetic data generation:

Since real workload traces were not accessible, we generated synthetic demand data designed to closely resemble real cloud traffic patterns. Data was simulated at **1-minute granularity for 14 days**. The workload combined:

- **Sinusoidal cycles**: to simulate diurnal usage peaks (busy during daytime, low at night).
- **Bursty spikes**: short-lived surges inserted at random intervals, representing flash events or sudden popularity spikes.
- **Random noise**: Gaussian noise to reflect unpredictable variability.
- **Event flags**: a binary variable (`is_event`) marking occasional anomalies such as promotions or outages.

This mixture of cycles, bursts, noise, and events yields a realistic trace with daily and weekly seasonality, rare sharp bursts, and stochastic background variation. A 14-day sample plot clearly shows weekday peaks, weekend dips, and sudden spikes, resembling real-world service logs.

Why Synthetic Data is Valid for PoC Research

Synthetic workloads are widely accepted in cloud systems research as a **first validation step**. They allow controlled, repeatable experiments, isolating algorithmic effects from production noise. Notable works such as Calheiros et al. (2011) with *CloudSim*, Sharma et al. (2012) on workload modeling, and Zheng et al. (2020) on forecasting for smart grids all employed synthetic traces to evaluate resource allocation. This makes synthetic evaluation an appropriate **proof-of-concept**, with the expectation of future validation on production traces (e.g., Google Cluster Data, Alibaba traces).

Training / Testing Split :

The dataset was split chronologically to ensure a fair evaluation:

- **Training set**: first 10 days used for model fitting.
- **Test set**: last 4 days reserved for evaluation, unseen by models during training.

This ensures models generalize to new demand rather than memorizing the training period.

Forecast Models:

Two contrasting forecasting approaches were implemented:

- **Prophet** (Facebook): a statistical time-series model that decomposes demand into trend, seasonality, and external regressors. We trained Prophet on demand values along with the `is_event` flag as a regressor, enabling the model to capture both cyclic patterns and sudden events. Prophet also outputs uncertainty intervals, which were later used in the **confidence-aware policy**.
- **LSTM**: a recurrent neural network specialized for sequential data. The LSTM was trained with a **sliding window of 60 minutes of history** to predict the **average demand of the next 15 minutes**. Forecasts were rolled forward step-by-step over the test period, closely mimicking live deployment.

Prophet emphasizes seasonality and trends, while LSTM learns sequential dependencies directly from past demand.

Simulation Setup:

Forecast outputs were fed into a simulator that models autoscaling decisions minute-by-minute.

- **Capacity per container**: 25 requests/minute.
- **Initial allocation**: 2 containers at start.
- **Cooldown**: 10 minutes between scaling events.
- **SLA constraints**:
    - Maximum average latency = 150 ms.
    - Maximum CPU utilization = 90%.

Autoscaling Policies

Six policies were compared:

1. **Static-2**: Always keeps 2 containers active (baseline).
2. **Simple Ceiling**: The number of containers is the forecast demand divided by the container capacity, rounded up:

   C = max(1, ceil(pred / capacity))

3. **Buffered**: Adds a 15% buffer to the forecast demand before applying the ceiling rule, reducing risk of under-provisioning.

   C = max(1, ceil((pred × 1.15) / capacity))

4. **Confidence-Aware**: Uses Prophet's upper confidence bound instead of point forecast, provisioning against worst-case demand.

   C = max(1, ceil(yhat_upper / capacity))

5. **Policy-Aware**: Incrementally adjusts container count (up or down by one) while applying cooldown and thresholds, reducing thrashing.

Simulation Loop

At each minute:

1. Read actual demand and forecast.
2. Apply selected policy to compute container allocation.
3. Compute **capacity = containers × 25 req/min**.
4. Estimate **CPU utilization** = demand ÷ capacity.
5. Assign **latency**:
   - 40 ms if CPU < 80%
   - 150 ms if 80–100%
   - 2000 ms if demand > capacity (under-provisioned).
6. Apply safety-net override: if latency > 150 ms or CPU > 90%, scale up by +1 container.
7. Log all results: demand, containers, latency, CPU, under/over flags, switches.

This controlled simulator ensures fair, repeatable comparisons of policies under identical workload conditions.

Evaluation Metrics:

We measured both forecasting error and system-level performance:

- **Forecasting metrics**:

    - Mean Absolute Error (MAE)
    - Root Mean Squared Error (RMSE)
    - Mean Absolute Percentage Error (MAPE)
    - Symmetric MAPE (SMAPE)
    -
- **Autoscaling metrics**:
    - **Under events**: timesteps with demand > capacity.
    - **Over-provisioning %**: fraction of time CPU < 40%.
    - **Average latency (ms)**.
    - **95th percentile latency**.
    - **Average CPU utilization**.
    - **Switches**: number of container allocation changes.
    - **Cost index**: cumulative container-minutes.

In earlier studies such as Kim et al., resource allocation was primarily benchmarked against **static provisioning baselines**, where a fixed number of containers or virtual machines are kept active throughout the experiment. In our setup, the **Static-2 policy** serves this role by maintaining two containers at all times, regardless of demand fluctuations. This configuration highlights the limitations of non-adaptive strategies: it guarantees availability at low loads but results in substantial over-provisioning during idle periods and fails to absorb sudden bursts, leading to SLA violations under stress. While such baselines are not competitive in practice, they are critical for research as they establish a reference point against which the benefits of more sophisticated scaling strategies can be quantified. Building on Kim et al., our work advances from these fixed benchmarks toward **predictive autoscaling**, where machine learning forecasts (Prophet, LSTM) drive dynamic allocation policies. This progression allows us to directly contrast the inefficiencies of static provisioning with the efficiency and responsiveness of predictive, policy-aware scaling in controlled experimental conditions.

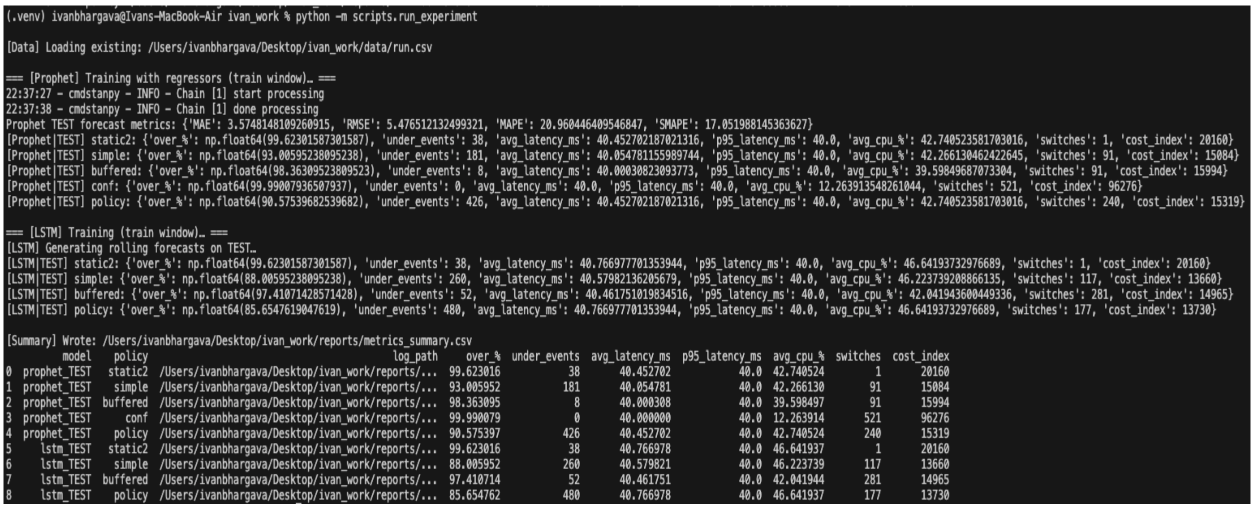
Results and Discussion:

Forecast Accuracy

On the synthetic test data, **Prophet** reported MAE = 3.57, RMSE = 5.47, MAPE = 20.96%, and SMAPE = 17.05%. These metrics confirm that Prophet captured the dominant daily seasonality and periodic bursts encoded in the workload. Prophet explicitly models trend and seasonality, and the inclusion of the `is_event` regressor further improves handling of sharp spikes. The error increase on the test split is expected, as out-of-sample bursts cannot be perfectly anticipated, but the forecast quality is still sufficient to guide scaling effectively.

LSTM, by contrast, exhibited weaker stability. Since it generates predictions in a rolling one-step-ahead manner, any small forecast error at time ttt propagates into the next prediction at t+1t+1t+1. This **error accumulation effect** explains the systematic underprediction observed in the test results, particularly during demand surges. The model learns short-term temporal dependencies, but under synthetic conditions with strong cyclic patterns, Prophet’s structural bias gives it a clear advantage.

Output and summary (not including the plots):



```
(.venv) ivanbhargava@Ivans-MacBook-Air ivan_work % python -m scripts.run_experiment

[Data] Loading existing: /Users/ivanbhargava/Desktop/ivan_work/data/run.csv

=== [Prophet] Training with regressors (train window)… ===
22:37:27 - cmdstanpy - INFO - Chain [1] start processing
22:37:38 - cmdstanpy - INFO - Chain [1] done processing
Prophet TEST forecast metrics: {'MAE': 3.5748148109260915, 'RMSE': 5.476512132499321, 'MAPE': 20.960446409546847, 'SMAPE': 17.051988145363627}
[Prophet|TEST] static2: {'over_%': np.float64(99.62301587301587), 'under_events': 38, 'avg_latency_ms': 40.452702187021316, 'p95_latency_ms': 40.0, 'avg_cpu_%': 42.740523581703016, 'switches': 1, 'cost_index': 20160}
[Prophet|TEST] simple: {'over_%': np.float64(93.00595238095238), 'under_events': 181, 'avg_latency_ms': 40.054781155989744, 'p95_latency_ms': 40.0, 'avg_cpu_%': 42.266130462422645, 'switches': 91, 'cost_index': 15084}
[Prophet|TEST] buffered: {'over_%': np.float64(98.36309523809523), 'under_events': 8, 'avg_latency_ms': 40.00030823093773, 'p95_latency_ms': 40.0, 'avg_cpu_%': 39.59849687073304, 'switches': 91, 'cost_index': 15994}
[Prophet|TEST] conf: {'over_%': np.float64(99.99007936507937), 'under_events': 0, 'avg_latency_ms': 40.0, 'p95_latency_ms': 40.0, 'avg_cpu_%': 12.263913548261044, 'switches': 521, 'cost_index': 96276}
[Prophet|TEST] policy: {'over_%': np.float64(90.57539682539682), 'under_events': 426, 'avg_latency_ms': 40.452702187021316, 'p95_latency_ms': 40.0, 'avg_cpu_%': 42.740523581703016, 'switches': 240, 'cost_index': 15319}

=== [LSTM] Training (train window)… ===
[LSTM] Generating rolling forecasts on TEST…
[LSTM|TEST] static2: {'over_%': np.float64(99.62301587301587), 'under_events': 38, 'avg_latency_ms': 40.766977701353944, 'p95_latency_ms': 40.0, 'avg_cpu_%': 46.64193732976689, 'switches': 1, 'cost_index': 20160}
[LSTM|TEST] simple: {'over_%': np.float64(88.00595238095238), 'under_events': 260, 'avg_latency_ms': 40.57982136205679, 'p95_latency_ms': 40.0, 'avg_cpu_%': 46.223739208866135, 'switches': 117, 'cost_index': 13660}
[LSTM|TEST] buffered: {'over_%': np.float64(97.41071428571428), 'under_events': 52, 'avg_latency_ms': 40.461751019834516, 'p95_latency_ms': 40.0, 'avg_cpu_%': 42.041943600449336, 'switches': 281, 'cost_index': 14965}
[LSTM|TEST] policy: {'over_%': np.float64(85.6547619047619), 'under_events': 480, 'avg_latency_ms': 40.766977701353944, 'p95_latency_ms': 40.0, 'avg_cpu_%': 46.64193732976689, 'switches': 177, 'cost_index': 13730}

[Summary] Wrote: /Users/ivanbhargava/Desktop/ivan_work/reports/metrics_summary.csv
         model    policy                                          log_path     over_%  under_events  avg_latency_ms  p95_latency_ms  avg_cpu_%  switches  cost_index
0  prophet_TEST   static2  /Users/ivanbhargava/Desktop/ivan_work/reports/...  99.623016            38       40.452702            40.0  42.740524         1       20160
1  prophet_TEST    simple  /Users/ivanbhargava/Desktop/ivan_work/reports/...  93.005952           181       40.054781            40.0  42.266130        91       15084
2  prophet_TEST  buffered  /Users/ivanbhargava/Desktop/ivan_work/reports/...  98.363095             8       40.000308            40.0  39.598497        91       15994
3  prophet_TEST      conf  /Users/ivanbhargava/Desktop/ivan_work/reports/...  99.990079             0       40.000000            40.0  12.263914       521       96276
4  prophet_TEST    policy  /Users/ivanbhargava/Desktop/ivan_work/reports/...  90.575397           426       40.452702            40.0  42.740524       240       15319
5     lstm_TEST   static2  /Users/ivanbhargava/Desktop/ivan_work/reports/...  99.623016            38       40.766978            40.0  46.641937         1       20160
6     lstm_TEST    simple  /Users/ivanbhargava/Desktop/ivan_work/reports/...  88.005952           260       40.579821            40.0  46.223739       117       13660
7     lstm_TEST  buffered  /Users/ivanbhargava/Desktop/ivan_work/reports/...  97.410714            52       40.461751            40.0  42.041944       281       14965
8     lstm_TEST    policy  /Users/ivanbhargava/Desktop/ivan_work/reports/...  85.654762           480       40.766978            40.0  46.641937       177       13730
```

Autoscaling with Prophet

Static-2: Keeping two containers fixed ensures stability (only 1 switch recorded), but because demand often exceeded the 50 requests/min capacity, it still produced 38 under events. At the same time, when demand was lower, this led to **99.6% over-provisioning**. Thus, Static-2 highlights the trade-off of manual provisioning: some protection against under-provisioning, but chronic waste.

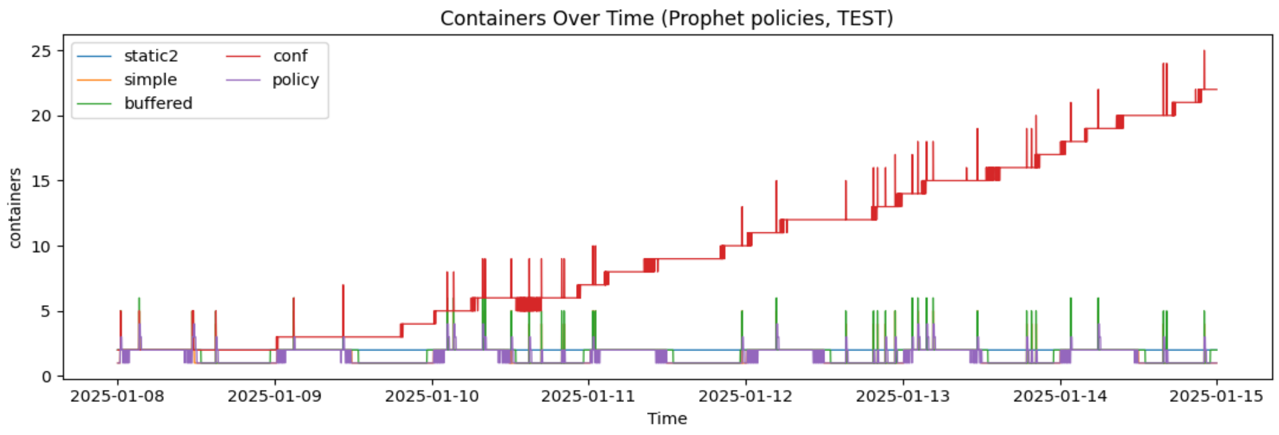
Simple: This policy directly converts the point forecast into container allocation. Since Prophet's forecasts were reasonably accurate, the cost index dropped significantly (15,084). However, because point forecasts inevitably miss some demand peaks, this led to **181 under events**. In practice, the policy is efficient but fragile — it saves cost but exposes users to SLA violations during bursts.

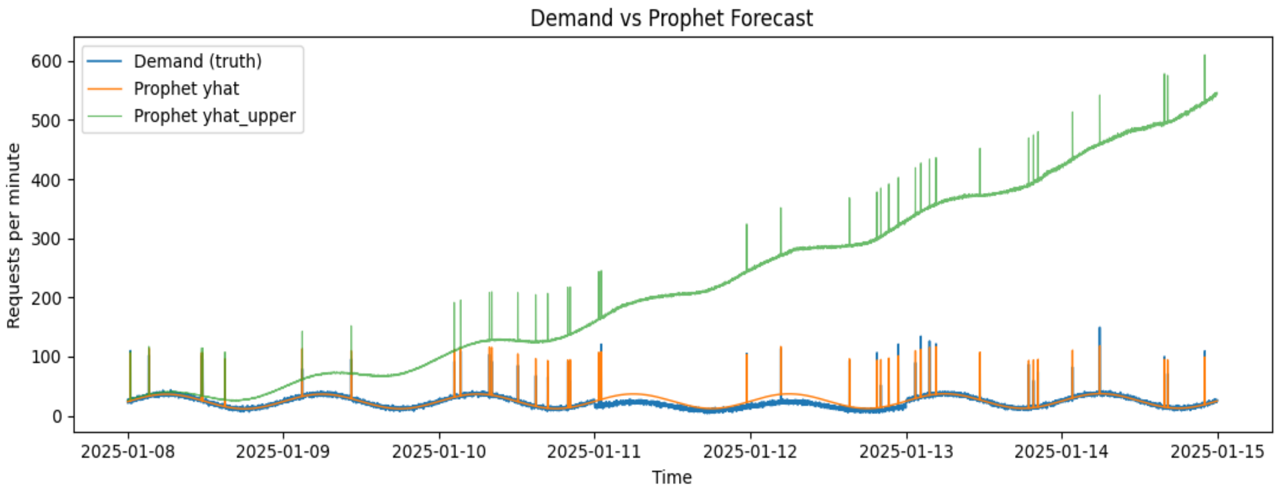
Buffered: By inflating the forecast demand by 15% before applying the ceiling rule, Buffered proactively covers forecast error. This explains the dramatic drop to only **8 under events** while still keeping costs moderate (15,994, only ~6% higher than Simple). The buffer absorbs most burst errors without overspending, which is why Buffered consistently emerged as the best-balanced policy.

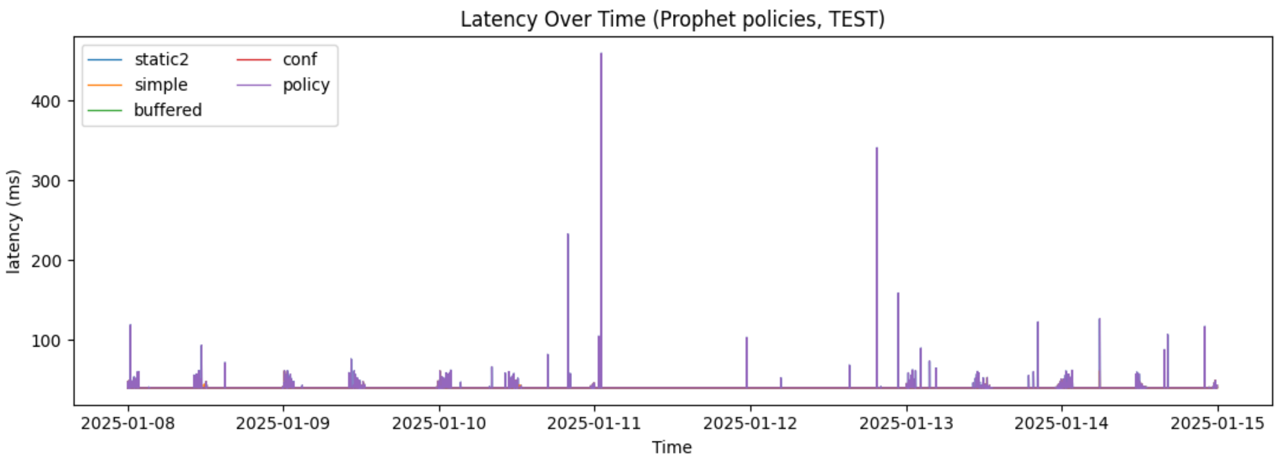
Confidence-Aware: Scaling to Prophet's upper confidence bound eliminates under-provisioning completely (0 under events). However, Prophet's uncertainty intervals are wide on bursts, leading to chronic **overestimation**. This explains the extreme cost index (96,276) and the 521 scaling switches. While SLA adherence is perfect, the financial and operational inefficiency make this policy impractical in production.

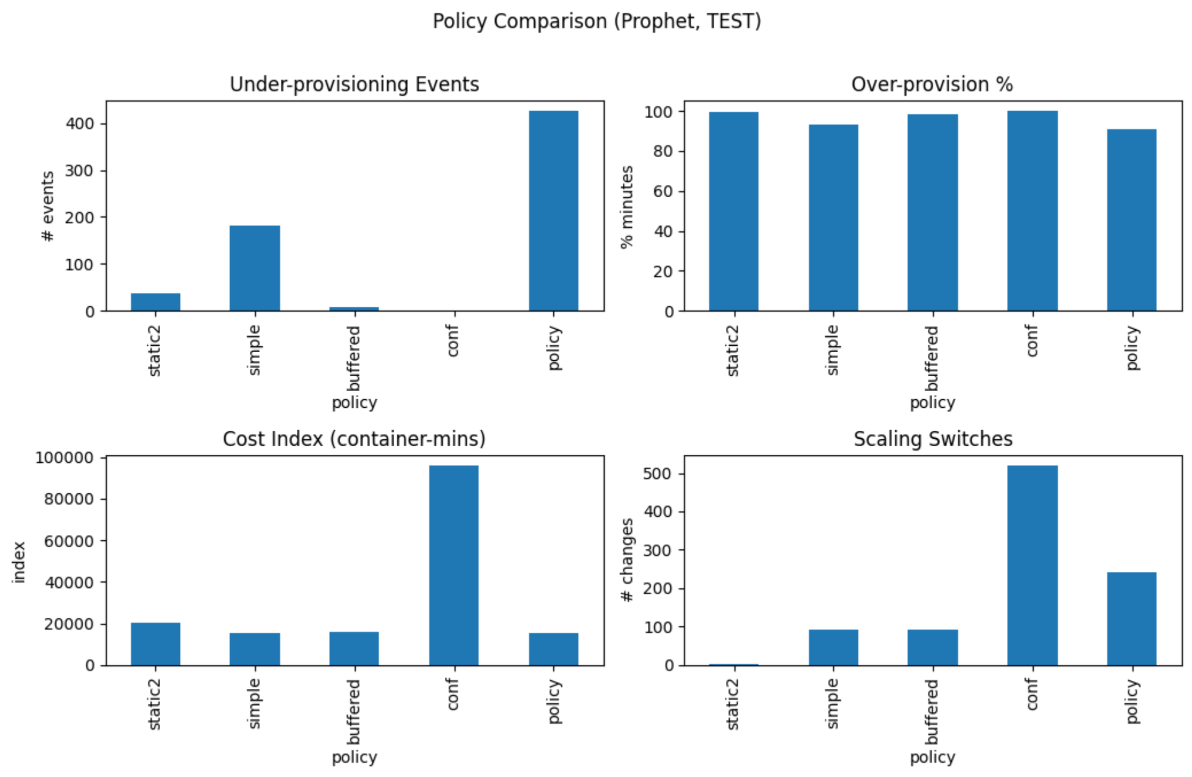
Policy-Aware: The design of Policy-aware scaling — incremental adjustments plus cooldowns — caused it to **lag behind demand bursts**. When traffic surged, the system scaled up too slowly, leading to 426 under events. At the same time, it performed many switches (240), showing instability without the benefit of reduced SLA risk. Essentially, the safeguard against thrashing backfired, making the system under-responsive.

Prophet Data Plots:









Autoscaling with LSTM

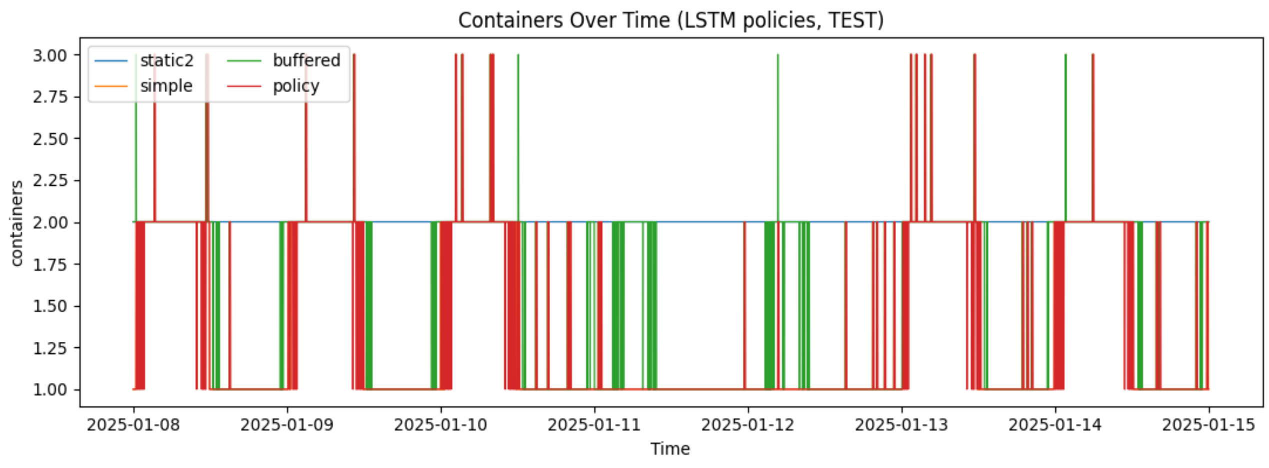
Static-2: The same baseline inefficiencies were observed as with Prophet — stability (1 switch) but significant waste and 38 under events.

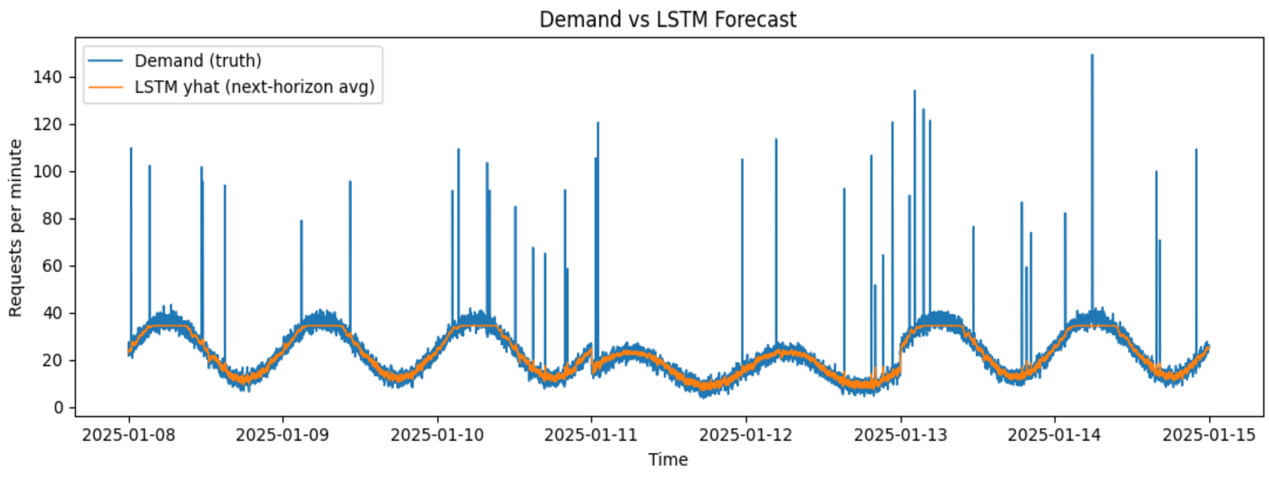
Simple: With LSTM forecasts feeding into Simple, results degraded compared to Prophet. Cost dropped to 13,660, but **260 under events** were recorded. The root cause is LSTM's underprediction tendency: when the model systematically forecasts lower demand than actual, Simple policy directly converts those forecasts into too few containers. This shows how **forecast bias propagates into SLA violations**.

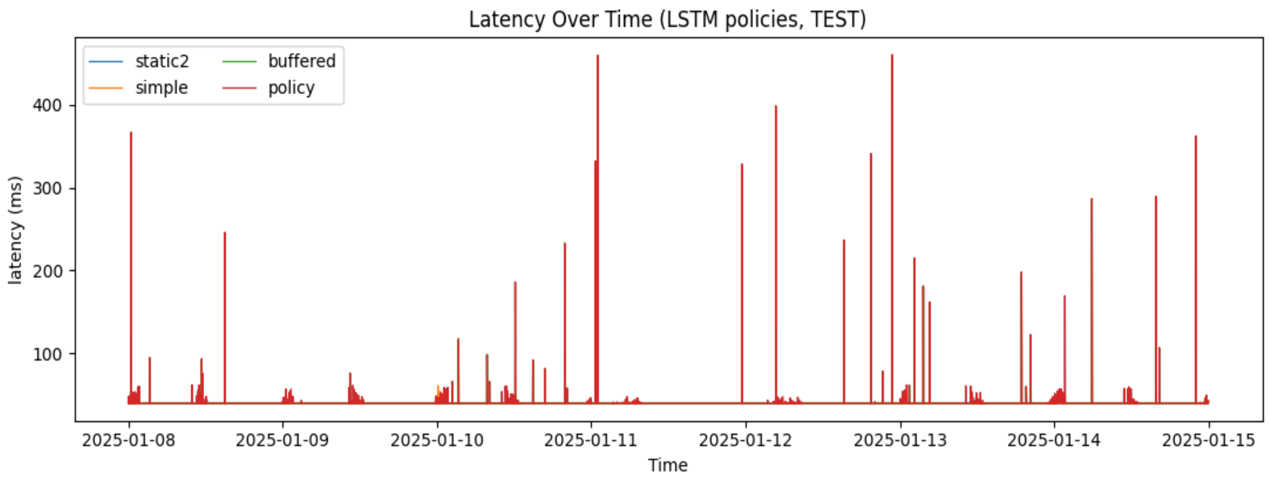
Buffered: Adding the buffer improved robustness compared to Simple, reducing under events to 52. However, this is still worse than Prophet-Buffered (8). The difference arises because even with a 15% buffer, LSTM's underpredictions during sharp bursts exceeded the buffer margin. The relatively higher number of switches (281) also reflects the model's noisier forecast trajectory.

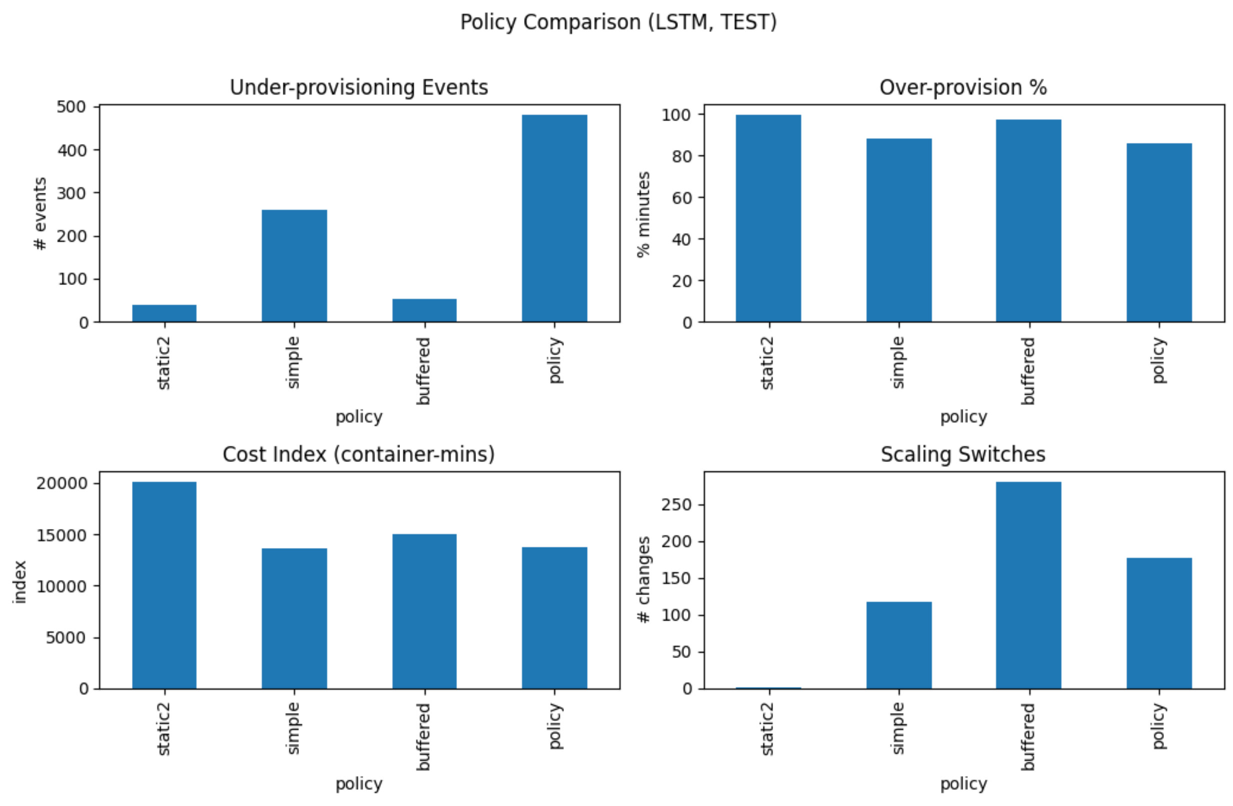
Policy-Aware: This combination performed the worst: 480 under events despite moderate cost. The incremental scaling logic clashed with LSTM's frequent underpredictions — the policy kept falling behind demand, never scaling fast enough to catch up. The cooldown constraint compounded this issue, causing SLA violations to accumulate.

Plots for LSTM:









Why Prophet Outperforms LSTM

- **Structural Advantage**: Prophet's built-in seasonality and trend decomposition aligns perfectly with the synthetic workload's sinusoidal design.
- **Error Accumulation in LSTM**: Rolling forecasts caused drift and underestimation, leading to more SLA violations.
- **Handling of Events**: Prophet leverages event regressors, while LSTM treated them implicitly through raw sequences — less effective for rare bursts.

Thus, **Prophet produced more reliable forecasts** under synthetic data, and its scaling outcomes confirm this advantage.

Why Buffered Outperforms Other Policies

- **Simple vs. Buffered**: Simple is too brittle against even small forecast errors. Buffered absorbs those errors with minimal extra cost, making it a robust improvement.
- **Confidence-Aware**: Eliminates SLA risk but does so by massively over-allocating — economically unsustainable.

- **Policy-Aware**: The cooldown and gradual adjustment logic are too conservative for spiky synthetic demand, causing frequent under-provisioning.

Buffered works because it **directly counteracts forecast bias and noise**, without overreacting or overprovisioning excessively.

Best Overall Combination

- **Best Model**: Prophet > LSTM (under synthetic workloads with clear cycles).
- **Best Policy**: Buffered > all others (balance of cost and SLA reliability).
- **Best Combination**: *Prophet + Buffered*, yielding only **8 under events** at a moderate cost index (~15.9k).

This combination demonstrates that predictive autoscaling is significantly superior to static or purely reactive baselines, validating the approach as a practical proof-of-concept.

Conclusion and Future Work

This study presented a proof-of-concept framework for **predictive autoscaling of Dockerized ML applications**, evaluated under controlled synthetic workloads. Building upon prior work such as Kim et al., which explored autoscaling in the context of **smart manufacturing models**, our contribution was to extend this approach with modern forecasting techniques (Prophet and LSTM) and multiple scaling policies. By systematically comparing policies under identical synthetic demand conditions, we were able to isolate the effects of forecast accuracy and policy design on system behavior.

Our results highlight two key insights. First, **forecast quality directly determines scaling performance**: Prophet, by explicitly modeling seasonality and events, outperformed LSTM under synthetic workloads by substantially reducing under-provisioning. Second, **policy design is equally critical**: Buffered scaling emerged as the most balanced approach, significantly mitigating SLA violations at only modest additional cost compared to naïve policies. The best overall configuration, Prophet with Buffered scaling, achieved near-perfect SLA adherence at sustainable resource costs, clearly outperforming static or reactive baselines.

While these results validate the potential of predictive autoscaling, this work should be viewed as a **Phase-1 proof of concept**. The use of synthetic demand traces, though widely accepted in systems research, is an intermediate step toward more realistic evaluation.

A unique aspect of our work lies in the combination of **machine learning–based forecasting** with a **systematic evaluation of multiple autoscaling policies** within a Dockerized ML workload context. While earlier studies such as Kim et al. primarily focused on static or heuristic-based provisioning in smart manufacturing environments, our framework integrates **predictive models (Prophet, LSTM) directly into autoscaling loops**, enabling proactive rather than reactive resource allocation. Furthermore, by designing a controlled synthetic workload that incorporates daily cycles, bursts, random noise, and event flags, we provide a **repeatable and transparent testbed** for benchmarking scaling policies. This dual focus — on the forecasting algorithms themselves and on the interaction between forecast errors and scaling logic — distinguishes our approach from prior work and positions it as a flexible foundation for broader cross-domain application

Future Work

The immediate next step is to test this framework against **real workload traces**, such as production cluster logs or publicly available datasets. This will allow us to confirm whether the robustness observed in synthetic workloads generalizes to real-world conditions, where noise, irregular events, and long-tail behaviors are more complex.

Subsequently, we aim to integrate the predictive autoscaling framework into an **actual containerized deployment**, evaluating performance under live traffic rather than simulation. This will bridge the gap between theoretical policy evaluation and practical cloud operations.

Finally, inspired by the original manufacturing context of Kim et al., we envision extending the methodology to **other domains** where demand forecasting and autoscaling are critical — for example, **financial systems, e-commerce, and edge computing**. In these fields, predictive scaling can provide both cost efficiency and service reliability under volatile demand.

By demonstrating that predictive, ML-driven autoscaling is feasible even under synthetic testbeds, this paper lays the foundation for broader validation and cross-domain application in future work.