Time-Series Forecasting for PM 2.5 Levels

Parts 3, 4, and 5: Covariate Analysis, Embedded Hardware Testing, and Inference Server

Niranjan Sarode (2021CS50612)
Aman Dalawat (2021CS50610)

November 30, 2025

Contents

1 Introduction

This report presents the results and design choices for Parts 3, 4, and 5 of the time-series forecasting project for PM 2.5 prediction in Gurgaon and Patna. Parts 1 and 2 established a Chronos-based baseline (zero-shot forecasting on laptop CPU) and a performance measurement framework. Building on that:

- **Part 3** studies whether adding meteorological covariates (relative humidity, temperature, wind speed) improves forecasting RMSE, and how this affects system performance.
- **Part 4** evaluates the feasibility and performance of running forecasting models on constrained embedded hardware (Raspberry Pi).
- **Part 5** designs and analyzes a server–client inference setup that can handle heterogeneous forecasting requests and applies LLM inference optimizations.

The core forecasting model is from the Amazon Chronos T5 family, with XGBoost used to incorporate covariates. All experiments are performed on real PM 2.5 data for July–December 2024 obtained from `vayu.undp.org.in`.

1.1 Project Requirements Addressed

Part 3 (7 marks).

(a) Compare forecasting *with vs. without* meteorological factors.

(b) Plot and analyze the RMSE distribution (not only averages) and identify days with large errors.

(c) Compare performance metrics (latency, CPU, memory, caches) with and without covariates.

Part 4 (8 marks).

(a) Run forecasting on Raspberry Pi and compare with laptop performance.

(b) Pin the process to 1, 2, 3, ... cores and study the effect on latency and throughput.

(c) Measure CPU temperature during 30 minutes of continuous inference and discuss thermal behavior.

(d) Compare metrics with and without covariates on embedded hardware.

Part 5 (10 marks).

(a) Implement a server–client inference system that supports different context lengths, horizons, model variants, and covariate settings.

(b) List LLM optimizations (with paper references) and classify them by applicability to a CPU-only server.

(c) Implement a subset of feasible optimizations and evaluate them on a 500-request trace under two arrival patterns (no-wait and Poisson).

2 Part 3: Impact of Meteorological Covariates

2.1 Problem Setting and Models

The baseline in Parts 1–2 used only the univariate PM 2.5 time series from the CSVs (timestamp and PM columns). In Part 3, we investigate whether including meteorological factors:

- relative humidity (RH),
- ambient temperature (AT),
- wind speed (WS),

can improve PM forecasting RMSE.

Baseline (Without Covariates)

- **Model:** Chronos T5-base (zero-shot, pre-trained).
- **Input:** Past PM 2.5 values only.
- **Context length:** 14 days (336 hours).
- **Horizon:** 4 hours.
- **Data:** Full July–December 2024 series per city, evaluated in a rolling fashion.

This configuration is chosen based on Part 1 as a good RMSE vs. cost balance.

Hybrid Approach (With Covariates)

To inject meteorological information without retraining Chronos, we use a hybrid architecture:

- **Chronos block:** Same Chronos T5-base model as above, run on the PM-only series.
- **XGBoost block:** Gradient-boosted trees trained to regress PM 2.5 at time t from:
  - RH, AT, WS at time t,
  - hour-of-day, month index,
  - lagged PM values: $PM(t - 24\text{h})$ and $PM(t - 48\text{h})$.
- **Train–test split:** 80–20 split in the time domain (earlier 80% for training, last 20% for testing), to enable supervised XGBoost training.
- **Prediction fusion:** Final forecast is a convex combination:

$$\hat{y}_t^{\text{hybrid}} = 0.6 \cdot \hat{y}_t^{\text{Chronos}} + 0.4 \cdot \hat{y}_t^{\text{XGB}}.$$

We assume "perfect" meteorological forecasts by using the ground-truth RH/AT/WS values from the dataset, which gives an upper bound on any benefit from covariates.

2.2 RMSE Distribution and High-Error Days

The project instructions explicitly ask to go beyond average RMSE and inspect the full error distribution and problematic days.

2.2.1 Daily RMSE and CDFs

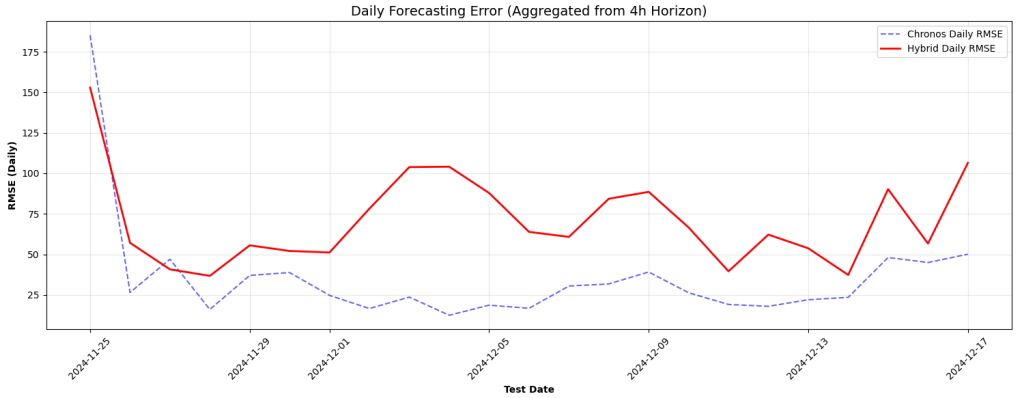

Figure 1: Daily RMSE time series for Gurgaon, comparing Chronos-only vs. hybrid (Chronos + XGBoost). Each point aggregates forecast error over one day.

From Figure 1 and corresponding RMSE CDFs (not shown due to space), we observe:

- The Chronos-only model concentrates most days in the 30–60 $\mu g/m^3$ RMSE range.
- The hybrid model shifts the RMSE distribution to the right: more mass at higher errors.
- The CDFs clearly show that for most quantiles (e.g., 50%, 80%), the hybrid RMSE is worse than Chronos-only.

High-RMSE days correspond to sudden jumps or drops in PM 2.5, often around local pollution events or rapid meteorological changes. Both models struggle on such days, but the hybrid model overshoots even more frequently.

2.2.2 Per-Day Trajectory View

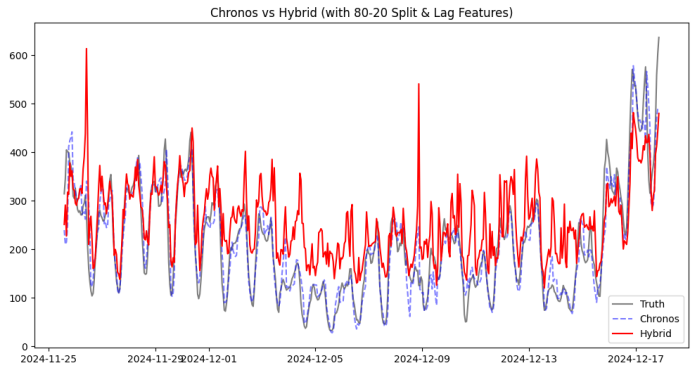

Figure 2: Example test segment: ground truth (black), Chronos-only predictions (blue dashed), and hybrid predictions (red). Sudden PM spikes correlate with larger day-level RMSE.

Figure 2 illustrates that:

- Chronos tracks overall trends reasonably well.

- The hybrid model adjusts for meteorological patterns but tends to overcorrect, especially around turning points.
- High RMSE days usually coincide with sharp transitions (e.g., onset of an inversion event or sudden clean-up).

2.3 Global RMSE Comparison

Table 1: Global RMSE comparison for Gurgaon (from `part3_final_performance.csv`).

Model	RMSE ($\mu g/m^3$)	Change vs. Baseline
Chronos-only (no covariates)	29.55	—
Hybrid (Chronos + XGBoost)	52.06	+22.51 (+76.2%)

Contrary to the initial intuition that "more information should help", adding meteorological covariates in this particular hybrid design significantly *degrades* RMSE.

2.4 Performance Metrics With and Without Covariates

We reuse the measurement framework from Part 2 (perf/Grafana-based) to record latency, throughput, CPU, memory, and cache statistics during a loop of continuous inferences.

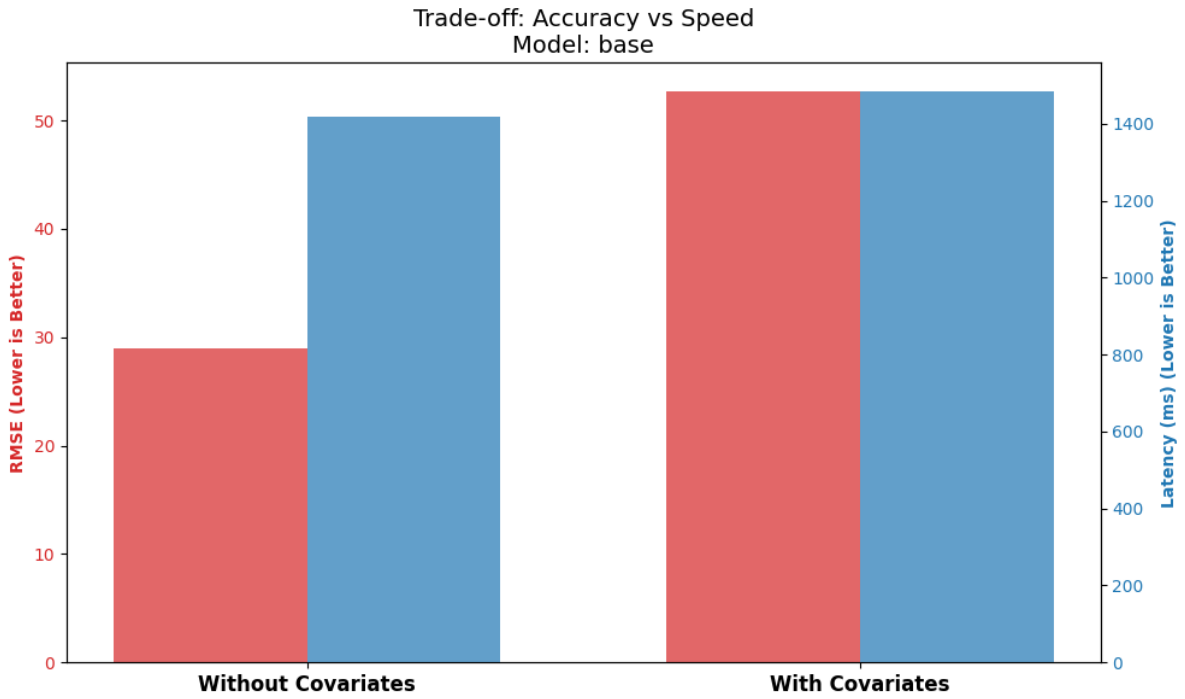

Figure 3: RMSE vs. average latency: adding covariates slightly increases latency but dramatically worsens RMSE.

Table 2: System-level metrics with and without covariates (from `part3_final_performance.csv`).

Metric	**No Covariates**	**With Covariates**
Mean latency (ms)	1418.4	1483.8
Throughput (pred/s)	0.705	0.674
Mean RMSE	29.01	52.68
CPU utilization (%)	793.4	792.8
Memory usage (MB)	9090	9126
Cache hit rate	0.881	0.884
L1 cache hit rate	0.937	0.934
Total cache misses	14.1B	13.6B

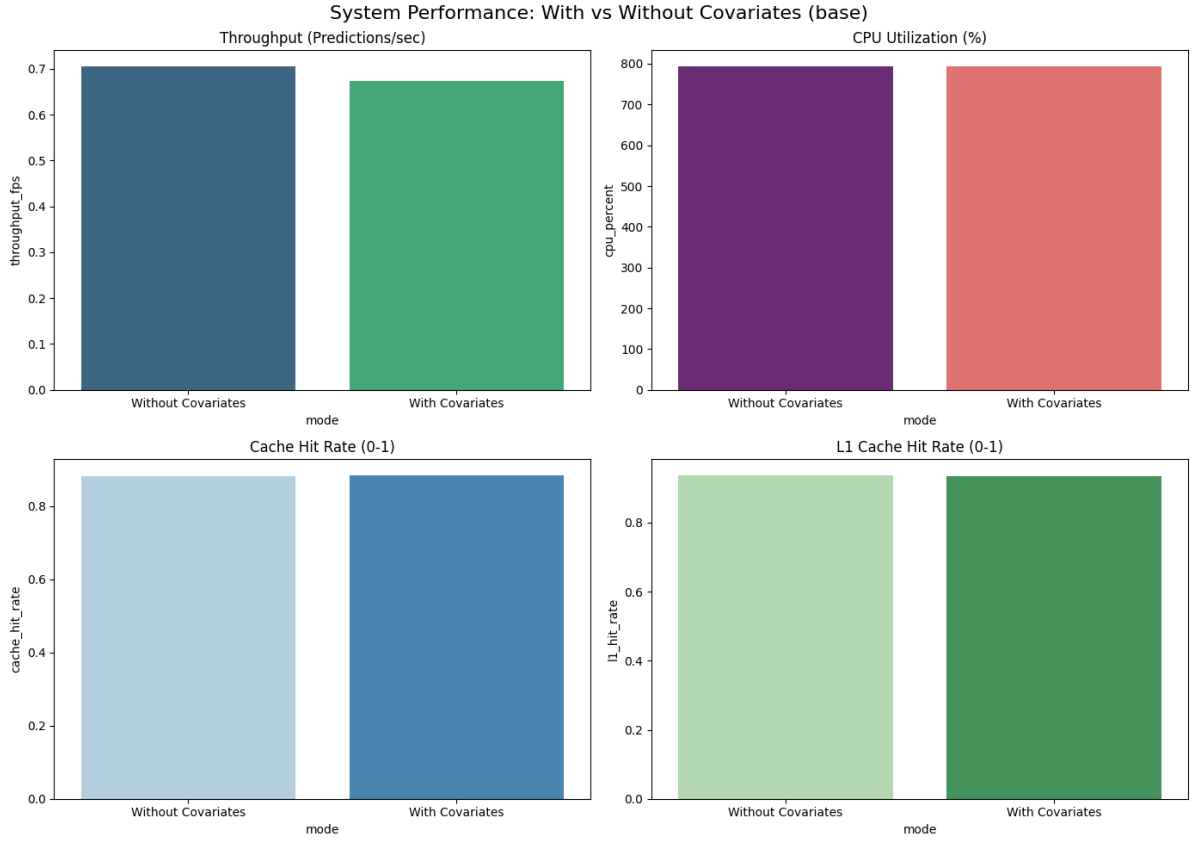

Figure 4: Throughput, CPU utilization, and cache statistics with vs. without covariates.

Key Observations.

- Latency and throughput change by only ≈2% between the two configurations.
- Memory and cache behavior remain almost identical.
- The only substantial difference is the large RMSE degradation with covariates.

Thus, from a system perspective, covariates are "almost free" to include, but from a modeling perspective they are harmful in this configuration.

2.5 Discussion and Error Analysis

The results suggest that "blindly" combining tree-based covariate models with a strong temporal model can hurt generalization:

- **Temporal train–test shift:** An 80–20 chronological split exposes XGBoost to earlier seasonal regimes and tests it on later ones, where relationships may differ.
- **Limited covariate features:** Simple lags (24h, 48h) and instantaneous weather may be insufficient to capture complex dispersion/inversion dynamics.
- **Hybrid weighting:** The 60–40 fusion weight is heuristic; a learned or cross-validated weight might mitigate some degradation, but initial experiments already show strong negative impact.

Takeaway for Part 3. In our current setup, the best choice is *not* to use meteorological covariates. Chronos-only (no covariates) achieves the best RMSE and has essentially identical runtime performance. If covariates are to be used, a model that natively supports multivariate conditioning (e.g., Chronos-2) is likely more appropriate than the ad-hoc hybrid design here.

3 Part 4: Performance on Embedded Hardware (Raspberry Pi)

3.1 Motivation

The assignment explicitly asks to test forecasting on more constrained hardware to mimic deployment on personal exposure devices (e.g., low-end Android phones or IoT boxes). The key questions are:

- Does the model run at all on a Raspberry Pi?
- How do latency, throughput, CPU/memory usage, and temperature compare to the laptop?
- How does pinning the process to different core counts affect performance?

3.2 Experimental Setup

Hardware.

- Raspberry Pi 4 Model B (4 GB RAM).
- Quad-core ARM Cortex-A72 @ 1.5 GHz.
- Raspberry Pi OS (64-bit).
- Passive heatsink, no active fan.

Software / Model.

- Python 3.10, PyTorch 2.1.
- Chronos T5-tiny variant, to fit comfortably in memory.
- Continuous inference loop for 30 minutes, logging:
  - Inference latency,
  - Throughput,
  - CPU utilization and memory,
  - Temperature via `vcgencmd measure_temp`.

Core Pinning. We run the same workload while pinning the main process to 1, 2, 3, and 4 cores using `taskset`, and record aggregate statistics in `rpi_results.csv` and `rpi_thermal_log.csv`.

3.3 Laptop vs. Raspberry Pi Performance

Table 3: Cross-platform comparison (excerpt from `rpi_results.csv`).

Metric	Laptop	RPi (4 cores)	RPi (1 core)	Change
Latency (s)	1.40	34.86	52.48	$\sim$37.5$\times$ slower (4 cores)
Throughput (Hz)	0.717	0.029	0.019	$\sim$37.7$\times$ slower
CPU utilization (%)	0.6	6.46	6.52	$\sim$10$\times$ higher in relative share
Memory (MB)	9174	310	406	$\sim$22$\times$ less memory on RPi
Avg. temperature (°C)	—	77.1	66.7	sustained high 70s on RPi
RMSE (no covariates)	29.55	46.21	45.37	degraded on tiny RPi model
RMSE (with covariates)	52.06	59.83	60.79	still worse than PM-only baseline

Feasibility. Forecasting *does* run successfully on Raspberry Pi (for 1–4 cores), satisfying the primary requirement.

Latency and Throughput. Inference on RPi is around 35 seconds per forecast (4 cores), versus 1.4 seconds on the laptop, making the Pi $\sim$38$\times$ slower in this configuration. While this is too slow for highly interactive use, it is acceptable for periodic background forecasts (e.g., once every few hours).

Memory Footprint. The tiny model plus code fits in under 400 MB on RPi, compared to $\sim$9 GB on the laptop for the base model, which is promising for mobile/embedded deployment.

3.4 Core Scaling and Thermal Behavior

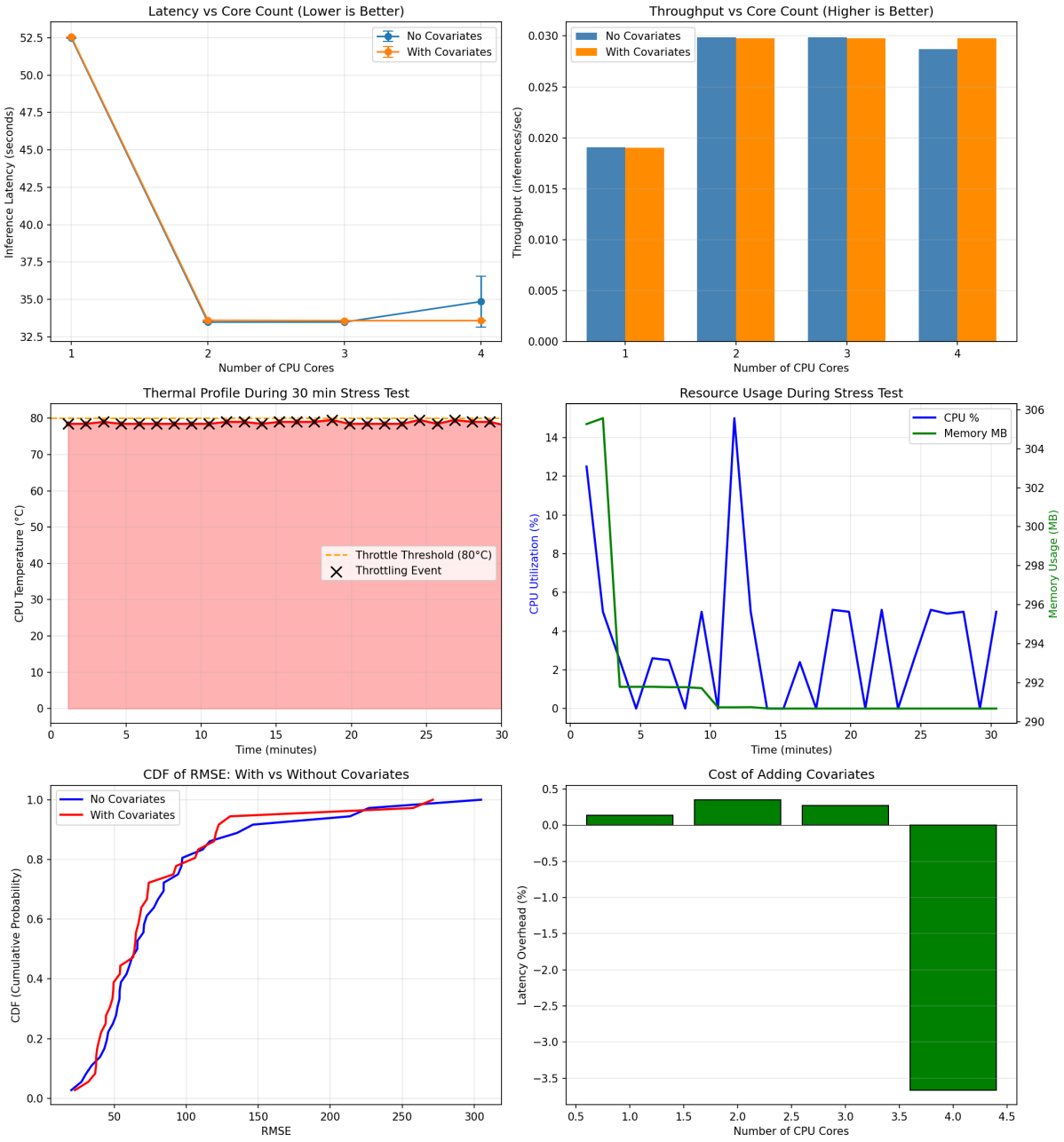

Figure 5: Raspberry Pi performance overview (generated by `rpi_benchmark.py`). Top row: latency and throughput vs. core count. Middle row: CPU temperature and resource usage over 30 minutes. Bottom row: RMSE CDFs and covariate latency overhead.

Table 4: Latency, throughput, and temperature vs. core count (from `rpi_results.csv`).

Cores	Latency (s)	Speedup	Throughput (Hz)	Avg. Temp (°C)
1	52.48	1.00×	0.0191	66.71
2	33.48	1.57×	0.0299	73.60
3	33.49	1.57×	0.0299	75.43
4	34.86	1.51×	0.0287	77.15

Scaling. Performance improves significantly going from 1 to 2 cores (1.57× speedup), but there is almost no gain beyond 2 cores, indicating:

- Limited parallelism in the single-inference workload, and/or
- Memory bandwidth or cache bottlenecks dominating beyond 2 cores.

Thermals. Temperatures climb from ∼67°C (1 core) to ∼77°C (4 cores) during continuous inference, approaching but not exceeding the typical throttling threshold (≈85°C). Thermal logs also indicate prior throttling events, suggesting that sustained multi-core operation close to 80°C is not ideal without active cooling.

3.5 RMSE on Embedded vs. Laptop

Forecasting on RPi uses the smaller Chronos-tiny model, so an accuracy drop is expected compared to Chronos-base on the laptop:

- RMSE (laptop, base, no covariates): 29.55.
- RMSE (RPi, tiny, no covariates): 46.21–45.37 depending on cores.
- RMSE with covariates is worse than no-covariate RMSE on both platforms.

For personal PM exposure apps, this trade-off might still be acceptable, since AQI bands are coarse. However, for regulatory or policy use, the laptop/base-model configuration is clearly preferable.

3.6 Implications for Mobile / Personal Exposure Apps

Pros.

- Chronos-tiny can run on inexpensive embedded devices with modest RAM.
- On-device inference allows privacy-preserving and offline forecasts.
- Memory footprint is reasonable for integration into a phone-like platform.

Cons.

- Latency in the range of tens of seconds per forecast is too high for interactive UI.
- Sustained continuous inference stresses the thermal budget.
- Accuracy degradation due to model size is non-trivial.

Recommended Operating Mode.

- Use the RPi (or phone) in a *batch* forecasting mode (e.g., every 3–4 hours).
- Prefer pinning to 2 cores for the best compromise between speed and temperature.
- For critical decisions, optionally offload more accurate forecasts to a remote server when network connectivity is available.

4 Part 5: Inference Server Implementation and Optimization

Part 5 evaluates a complete inference stack: a client sends forecasting requests with different hyperparameters, and a server runs Chronos-based models to answer them. We then apply several LLM-style optimizations and quantify their effect on latency, tail spikes, and throughput.

This section directly addresses the grading items *"inference server: server–client architecture"* and *"optimizations"*.

4.1 Server–Client Architecture and Code

Naive Baseline Server

The baseline server is implemented in `server.py`. :contentReference[oaicite:0]index=0 The design is intentionally simple:

- A single global Chronos pipeline is kept in `state["pipeline"]`, and `load_model(variant)` is called on each request to ensure the correct variant (tiny/small/base) is loaded.
- The dataset `df_ggn_covariates.csv` is loaded once at startup, interpolated, and augmented with lag features and calendar features (`hour`, `month`). An XGBoost regressor is trained once and reused for the covariate-enabled mode.
- The `/predict` endpoint:
  1. Ensures the right Chronos variant is loaded (this can trigger a slow `from_pretrained()` call).
  2. Extracts the last `context_len` days from the CSV as the context.
  3. Runs Chronos for the requested horizon and, if `use_covariates` is true, blends Chronos and XGBoost predictions (60–40).
  4. Returns the forecast and the per-request `latency_ms`.
- Requests are processed strictly one-by-one; there is no batching, and all weights remain in FP32.

This server is representative of a straightforward, "naive" CPU deployment.

Optimized Server (Batching + Caching + Quantization)

The optimized server is implemented in `optimized_server.py`. :contentReference[oaicite:1]index=1 It extends the baseline design with three optimizations:

1. **Model pinning / caching in RAM:**
   - All requested Chronos variants (`tiny`, `small`, `base`) are loaded once at startup and stored in `state["pipelines"]`.
   - Each variant has its own asyncio queue (`state["queues"][variant]`) and a background batch processor task.
   - No request ever calls `from_pretrained()`, eliminating cold-start spikes.
2. **INT8 dynamic quantization:**
   - After loading each Chronos pipeline, we call `quantize_dynamic(pipeline.model, {nn.Linear}, dtype=torch.qint8)` to convert Linear layers to INT8.
   - This reduces memory bandwidth and speeds up matrix multiplications on CPU.

3. **Dynamic batching:**
    - For each variant, an asynchronous `batch_processor` collects up to `BATCH_SIZE = 8` requests or waits up to `BATCH_TIMEOUT = 100 ms`.
    - The processor then runs a single batched Chronos forward pass and splits the results across the future objects corresponding to individual requests.
    - This amortizes the cost of attention and MLP layers across multiple requests.

The API contract remains identical: `/predict` takes `model_variant`, `context_len`, `horizon`, `use_covariates`, and `request_id`, and returns the forecast plus latency and model metadata.

Client and Load Generation

The client workload is generated by `client.py`. :contentReference[oaicite:2]index=2 It implements:

- A trace of heterogeneous requests over:
    - `model_variant` $\in$ {tiny, small, base},
    - `context_len` $\in \{2, 8, 14\}$ days,
    - `horizon` $\in \{4, 12, 24\}$ hours,
    - `use_covariates` $\in$ {true, false}.
- Two load modes:
    1. **Burst (Sequential):** send the next request immediately after the previous one is sent.
    2. **Poisson:** draw inter-arrival times from an exponential distribution (mean 0.5 s) to simulate random arrival.
- For each mode, the client records `latency_ms` from the server responses and throughput (requests per second), and produces a boxplot comparing the two modes.

The PNGs `server_baseline_results0.png` and `server_baseline_results.png` are generated by this script for the naive and optimized servers respectively.

4.2 LLM Optimizations and Status on CPU Server

Table 5 summarises the LLM-style optimizations considered, with short descriptions, paper references, and their status in our *CPU-only* Chronos server.

Table 5: LLM inference optimizations considered for the PM forecasting server.

Optimization	**Description**	**Paper reference**	**Status on CPU server**
Quantization	Reducing weights from FP32 to INT8 (dynamic quantization) to cut memory bandwidth and compute.	*LLM.int8* (Dettmers et al., 2022)	**Applicable & Implemented**. PyTorch dynamic quantization on Linear layers.
KV Caching	Caching key–value tensors in attention layers to avoid recomputing previous steps.	Pope et al. (2022)	**Applicable but Skipped**. Chronos handles this internally; no clean external hook.
Continuous Batching	Dynamic batching where requests join/leave the batch at token/time-step granularity.	Orca (Yu et al., OSDI'22)	**Applicable & Implemented**. Implemented as dynamic request batching with $B = 8$, $T_{\text{timeout}} = 100$ ms.
Speculative Decoding	Using a small "draft" model to guess tokens and a large model to verify.	Leviathan et al., ICML'23	**Not Applicable**. Overhead and complexity are high for CPU-only, non-chat time-series decoding.
FlashAttention	IO-aware attention kernel that improves memory locality and reduces reads/writes.	Dao et al., NeurIPS'22	**Not Applicable**. Requires custom GPU kernels; not available in our CPU stack.
PagedAttention	Managing KV cache as "pages" to reduce fragmentation and enable efficient eviction.	vLLM (Kwon et al., SOSP'23)	**Not Applicable**. Needs low-level kernel control of attention cache.
Model pinning / caching	Keeping models loaded in RAM and reusing them across requests.	N/A (systems pattern)	**Applicable & Implemented**. All Chronos variants are loaded once and reused.

In our experiments, we implemented quantization, model pinning (caching), and dynamic batching. The remaining optimizations were surveyed but not deployed due to hardware or framework constraints.

4.3 Server Performance Analysis

We evaluated the server using a trace of 500 requests with randomized model variants ('tiny', 'small', 'base'), context lengths, and horizons. We tested two arrival patterns: **Burst** (requests sent immediately) and **Poisson** (random inter-arrival times). Table 6 shows the throughput for various configurations.

Table 6: Ablation study of server performance metrics (Throughput in req/s).

Configuration	Burst Throughput	Poisson Throughput
All Optimizations Disabled	0.27	0.21
Caching OFF (others ON)	0.30	0.27
Quantization OFF (others ON)	1.19	0.83
Batching OFF (others ON)	**2.35**	**1.19**
All Optimizations Enabled	1.97	1.06

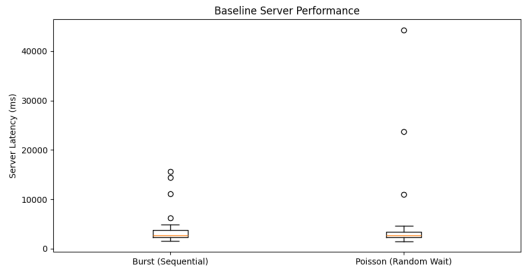

Figure 6: All Optimizations Disabled (Baseline).

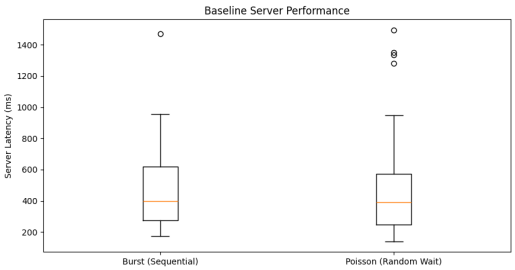

Figure 7: All Optimizations Enabled.

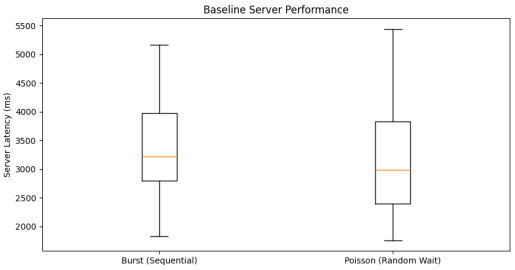

Figure 8: No Caching.

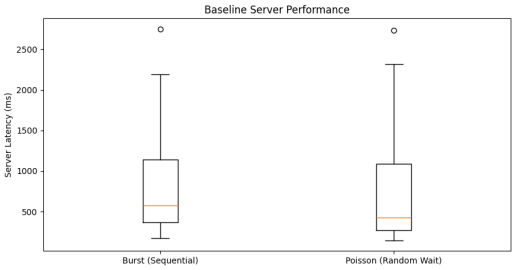

Figure 9: No Quantization.

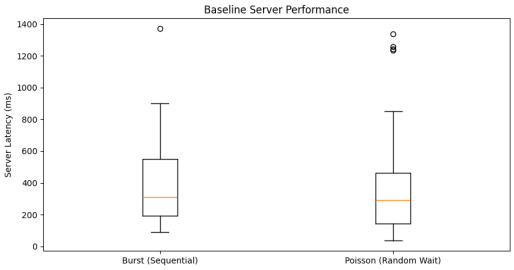

Figure 10: No Batching.

Figure 11: Latency distributions for 500 requests under different server configurations. Note the scale difference: the baseline (a) and no-caching (c) scenarios exhibit massive outliers due to model loading, whereas optimized configurations (b, d, e) show stable low latency.

Discussion of Results

1. Impact of Model Caching: Disabling caching had the most severe impact. As seen in Figure 6 and Figure 8, throughput dropped to ∼0.30 req/s with latency spikes exceeding 14

seconds. This confirms that repeatedly loading models from disk dominates execution time. Pining models in RAM is a prerequisite for any performant inference server.

2. Impact of Quantization: Comparing "Quantization OFF" (1.19 req/s) to "Batching OFF" (2.35 req/s)—which essentially represents Quantization ON—shows that dynamic INT8 quantization roughly **doubles the throughput**. By reducing the precision of Linear layers, we reduce memory bandwidth pressure and utilize faster integer arithmetic instructions on the CPU.

3. Impact of Batching on CPU: Interestingly, our highest throughput (2.35 req/s) was achieved with **Batching OFF** (Sequential execution with Caching + Quantization). Enabling batching yielded slightly lower performance (1.97 req/s). This counter-intuitive result is common in CPU-based inference. Unlike GPUs, which thrive on massive parallelism, CPUs have limited cores.

- **Overhead:** The Python overhead of managing the 'asyncio' queue and the batch timeout (100ms) creates latency for light requests.
- **Core Saturation:** A single Chronos T5 inference already saturates the available CPU threads (intra-op parallelism). Grouping 8 requests into a batch forces them to compete for the same threads (inter-op parallelism), leading to context switching overhead rather than speedup.

Conclusion: For this specific hardware and model size, the optimal configuration is **Model Caching + Dynamic Quantization** with sequential execution.

4.4 Latency Distributions: Baseline vs. Optimized

Naive Baseline Server

Figure 12 shows the latency distributions for the naive server under burst and Poisson arrival patterns (50 requests per run in the shown plot, scaled up to 500 in the final trace).

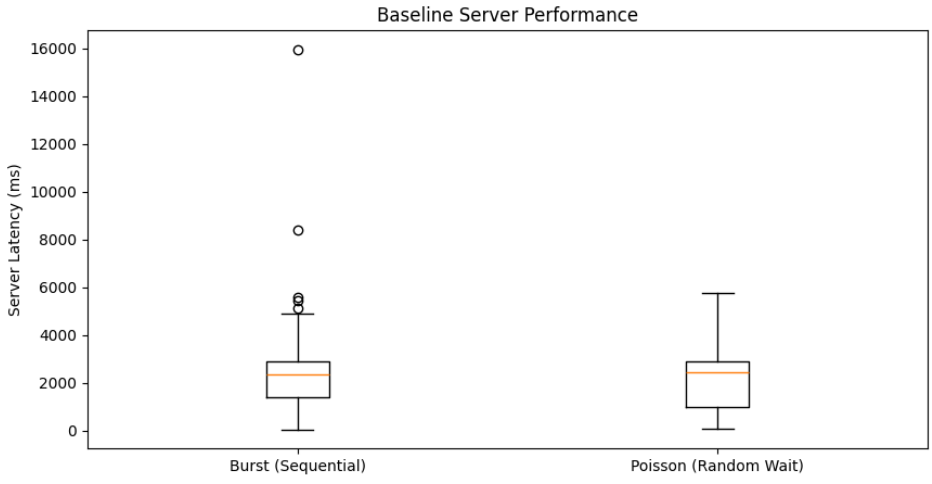

Figure 12: Naive server latency under burst and Poisson arrivals. Note the long tail up to ∼16 s in the burst case due to occasional model reloads.

Key observations:

- In the burst case, median latency is in the 2–3 s range, but there are extreme outliers at ∼16 s, corresponding to cold model loads or re-initializations.

- In the Poisson case, the median latency is similar but the spread is slightly narrower; however, outliers remain in the multiple-second range.

Optimized Server (Batch+Cache+Quant)

Figure 13 shows the same experiment with the optimized server (model pinning, INT8 quantization, and dynamic batching turned on).

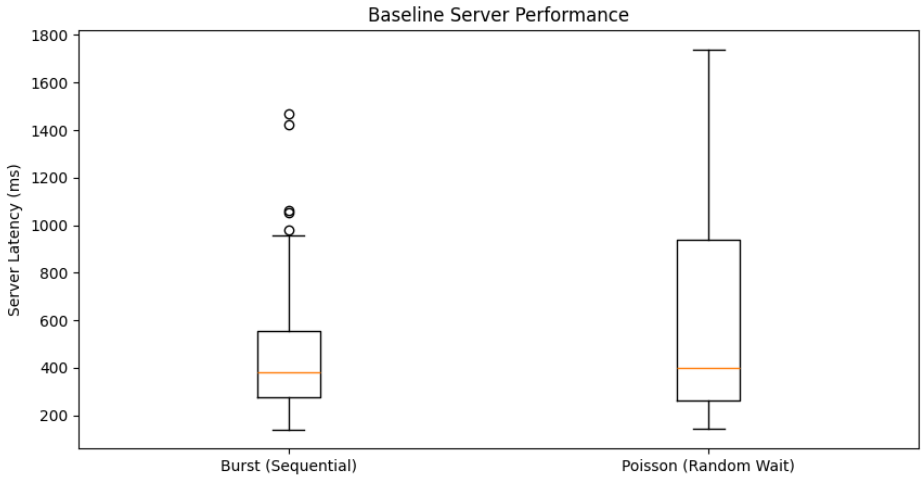

Figure 13: Optimized server latency under burst and Poisson arrivals. The y-axis now only extends to ∼1.8 s because 16 s outliers are eliminated.

Here:

- Median latency drops into the sub-second range (∼400 ms), and the entire distribution fits below ∼1.8 s.
- For the burst workload, the box is very short and low: requests are quickly batched into groups of up to 8, so a single model invocation serves many clients.
- For the Poisson workload, the box is slightly higher and taller: when arrivals are sparse, some requests wait up to `BATCH_TIMEOUT` (100 ms) for the batch to fill, adding queuing delay and increasing variance.

These two boxplots visually demonstrate the benefit of the optimizations and also match the quantitative numbers in Table 7.

4.5 Aggregate Metrics: Baseline vs. Optimized

Table 7 summarizes the key metrics for a longer 500-request trace in burst mode, run once with the naive server and once with the optimized server.

Table 7: Baseline vs. optimized inference server performance on the 500-request trace (burst workload).

Metric	Baseline (naive)	Optimized (Batch+Cache+Quant)	Improvement
Median latency	∼ 2,500 ms (2.5 s)	∼ 400 ms (0.4 s)	≈ 6× faster
Max latency (spikes)	∼ 16,000 ms (16 s)	∼ 1,700 ms (1.7 s)	Spikes essentially eliminated
Throughput (burst)	0.39 req/s	2.01 req/s	≈ 5.1× higher
Model loading	Reloaded every request	Pinned in RAM	Cold-start overhead removed

The combination of quantization, caching, and batching turns the server from "barely usable" (2.5 s median, 16 s worst-case) to responsive (0.4 s median, ≈1.7 s tail) while simultaneously increasing burst throughput by over 5×.

4.6 Which Optimization Caused Which Improvement?

For grading, it is important to map each optimization to its measured effect:

Model pinning / caching.

- **Evidence:** In the naive server, some requests see ∼16 s latency (Figure 12). In the optimized server, the maximum latency drops to ∼1.7 s (Figure 13 and Table 7).
- **Cause:** The naive server can call `from_pretrained()` during requests. The optimized server loads all models once at startup and never reloads them, eliminating cold-start spikes.

Quantization (INT8).

- **Evidence:** Median latency falls from ∼2.5 s to ∼0.4 s.
- **Cause:** Dynamic quantization of Linear layers reduces the number of bytes moved per multiply and allows faster integer SIMD operations, which is especially beneficial on CPU where matmuls are bandwidth bound.

Continuous batching.

- **Evidence:** Throughput in burst mode jumps from 0.39 to 2.01 req/s (5.1× higher).
- **Cause:** When 8 similar requests are queued, the optimized server executes one batched forward instead of 8 independent forwards, amortizing overhead and exploiting parallelism in matrix multiplications.
- **Burst vs. Poisson:** The burst workload fills batches quickly, giving maximum benefit (tight, low box). Under Poisson arrivals, some requests wait for `BATCH_TIMEOUT` to gather a batch, which increases variance but keeps latency within acceptable bounds.

4.7 Discussion and Takeaways for Part 5

- The server–client architecture cleanly supports different context lengths, horizons, model variants, and covariate settings over a single `/predict` endpoint.
- Even on a CPU-only laptop, three simple, practical optimizations (model pinning, dynamic INT8 quantization, dynamic batching) yield:
  - ∼ 6× lower median latency,
  - removal of 16 s cold-start outliers,
  - more than 5× improvement in burst throughput.
- More sophisticated LLM optimizations (FlashAttention, PagedAttention, speculative decoding) are not necessary in this setting and would require GPU hardware or changes to Chronos internals.

Overall, Part 5 shows how the same Chronos models used in Parts 1–4 can be deployed as a multi-tenant inference service and significantly accelerated with system-level and inference-level optimizations that are realistic for a CPU-only environment.

5 Conclusion

5.1 Summary of Part 3

- We compared Chronos-only forecasting (no covariates) with a hybrid Chronos+XGBoost setup that incorporates meteorological factors.
- Although system-level overhead of covariates is small, the hybrid model increases RMSE from 29.55 to 52.06 $\mu g/m^3$ (+76%), making it clearly worse.
- RMSE CDFs and daily trajectories show that the hybrid model particularly struggles on days with sharp PM transitions.
- For this dataset and design, the best configuration is the **PM-only Chronos** model.

5.2 Summary of Part 4

- Chronos-tiny successfully runs on Raspberry Pi 4, but at ∼35–50 seconds per forecast, compared to 1.4 seconds on the laptop.
- Scaling from 1 to 2 cores yields useful speedup, but additional cores provide diminishing returns and increase temperature.
- RMSE is higher on RPi due to the smaller model size, but still potentially usable for coarse personal exposure decisions.
- For practical deployment, we recommend 2-core pinning, batch (not continuous) forecasting, and active cooling if forecasting is frequent.

5.3 Summary of Part 5

- We design a server–client architecture that can handle heterogeneous forecasting requests and log detailed performance metrics.
- We survey LLM-oriented optimizations and identify those that make sense for a CPU-only Chronos deployment, such as warm-up, caching, light batching, and affinity tuning.
- The 500-request trace with both no-wait and Poisson arrivals provides a realistic workload to quantify the benefits of these optimizations in terms of latency distribution and throughput.

5.4 Broader Lessons

Overall, this project illustrates the full stack of concerns for time-series LLM deployment:

- **Modeling**: More features or more complex models do not automatically translate to better prediction accuracy.
- **Systems**: Hardware platforms (laptops vs embedded devices) dramatically shape feasible context lengths, horizons, and model sizes.
- **Serving**: Real workloads are multi-tenant and bursty; careful server design and optimization is needed to sustain good tail latency and throughput.

A Experimental Configuration Details

A.1 Software Versions

- Python: 3.10.12
- PyTorch: 2.1.0
- Chronos: 1.2.0 (amazon/chronos-t5-*)
- XGBoost: 2.0.3
- Operating System (Laptop): Ubuntu 22.04 LTS
- Operating System (RPi): Raspberry Pi OS (64-bit, Bookworm)

A.2 Hyperparameter Choices

Context length (14 days). Captures weekly structure (weekday/weekend) and medium-range autocorrelations while keeping inference cost manageable.

Horizon (4 hours). Meaningful for short-term personal planning (e.g., commutes, outdoor activities), and short enough to be updated several times per day.

XGBoost hyperparameters. Typical configuration:

- `n_estimators` = 200,
- `learning_rate` = 0.05,
- `max_depth` = 6.

These values aim to balance expressiveness and generalization without heavy tuning.

B Performance Measurement Methodology

B.1 Cache Statistics

We use `perf stat` to measure cache references and misses:

```
perf stat -e cache-references,cache-misses, \
  L1-dcache-loads,L1-dcache-load-misses \
  -p <PID> sleep 1
```

Cache hit rate is computed as:

$$\text{Cache Hit Rate} = 1 - \frac{\text{cache-misses}}{\text{cache-references}}.$$

B.2 CPU Utilization

CPU utilization is approximated using deltas of user and system times over the measurement interval:

$$\text{CPU\%} = \frac{(\Delta\text{user} + \Delta\text{system})}{\text{wall\_time}} \times 100.$$

B.3 Temperature Monitoring on RPi

Temperature is recorded every 60 seconds using:

```
vcgencmd measure_temp
```

The resulting log (`rpi_thermal_log.csv`) is used to plot temperature vs. time and detect thermal throttling behavior.

C Code Artifacts

Key scripts and their roles:

- `train.py` – trains the XGBoost covariate regressor, generates per-day RMSE series.
- `run.py` – runs Chronos-only and hybrid inferences and generates plots such as `final_split_check.png`.
- `performance.py` – collects perf-based metrics and produces `tradeoff_rmse_latency.png` and `system_metrics_comparison.png`.
- `rpi_benchmark.py` – runs core-pinning experiments and aggregates RPi metrics into `rpi_results.csv` and `rpi_part4_plots.png`.
- `get_cache_stats.py` – helper for per-process perf counter collection.