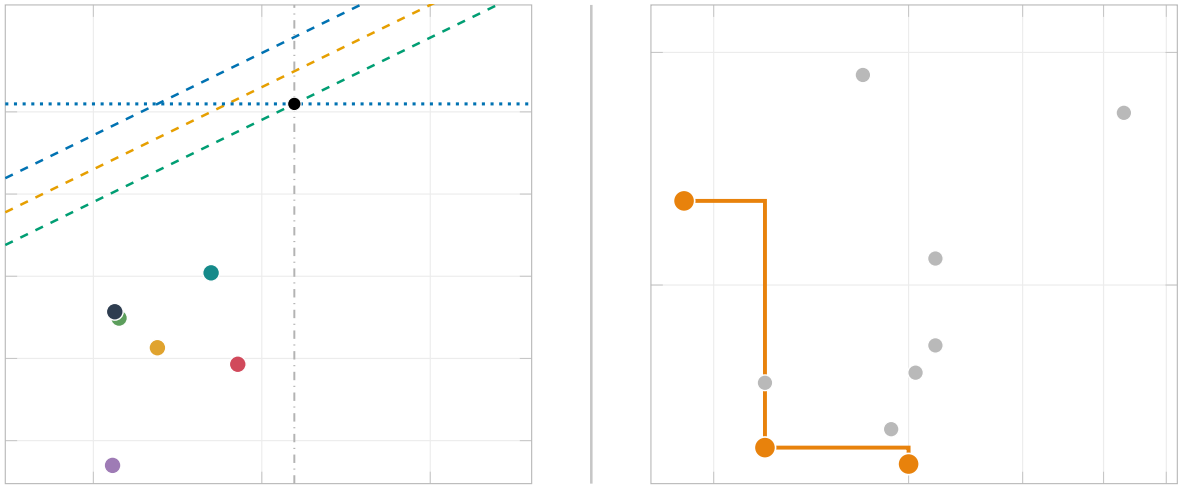
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Cost-Latency Benchmarking for Time-Series Forecasting

Hardware-Aware Evaluation of Deep Learning Architectures for Financial Planning

Master's thesis in Computer science and engineering

GUSTAF ASPLUND
FELIX BJERHEM ARONSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Cost-Latency Benchmarking for Time-Series Forecasting

Hardware-Aware Evaluation of Deep Learning Architectures for Financial Planning

GUSTAF ASPLUND
FELIX BJERHEM ARONSSON

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Cost-Latency Benchmarking for Time-Series Forecasting
Hardware-Aware Evaluation of Deep Learning Architectures for Financial Planning
GUSTAF ASPLUND
FELIX BJERHEM ARONSSON

Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Advisor: Tomas Harryson, Visma Spiris AB
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The two analytical lenses of this thesis, an inference roofline (left), placing the benchmarked forecasting models against the memory-bandwidth and compute ceilings of the hardware, and a cost–latency Pareto frontier (right) over the evaluated cloud instances.

Typeset in LaTeX
Gothenburg, Sweden 2026

Cost-Latency Benchmarking for Time-Series Forecasting
Hardware-Aware Evaluation of Deep Learning Architectures for Financial Planning
GUSTAF ASPLUND
FELIX BJERHEM ARONSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Enterprise financial planning is increasingly moving from statistical forecasting toward deep learning, which captures more complex patterns at the product level but raises the cost of serving forecasts. The hardware for these workloads is often chosen through heuristics rather than measurement, and existing serving research has focused on computer vision and language rather than time-series forecasting.

This thesis develops a workload-aware benchmarking framework that characterizes the cost-latency trade-offs of deep learning forecasting architectures across commodity cloud hardware. Six forecasting models spanning distinct computational classes were benchmarked on eleven Azure instances, examining how the operational intensity of each architecture sits against the roofline limits of the hardware, how far cost-latency behavior on real enterprise resource planning data diverges from simpler synthetic data, and how a Pareto analysis can guide the choice of a hardware and model pair under a given latency constraint.

Operational intensity stayed within a narrow band across the tested models, yet the point at which a workload becomes compute- or memory-bound shifted with the hardware, so the same model could be bound differently from one machine to the next. The cost-latency outcome thus depends on the model and hardware as a pair rather than on the architecture alone, and this shows in the provisioning results. Neither the largest CPU nor the newest accelerator reliably improved serving performance and an older, lower-tier GPU instance offered the best overall balance for the workloads tested.

The framework itself, rather than any single figure, is the more transferable result, and the comparison with synthetic data suggests it can stand in as a cheap first pass for narrowing the hardware search before real-data runs settle a final deployment.

Keywords: Time-series forecasting, deep learning, benchmarking, roofline model, cost-latency trade-off, cloud computing, inference serving, hardware selection, operational intensity, enterprise financial planning.

Acknowledgements

The path to completing this thesis has involved many difficult challenges, but also provided us with a lot of valuable insights and personal growth. We would like to thank Ahmed Ali-Eldin Hassan for supervision and guidance during this project.

This research for this thesis was funded by Spiris, and we gratefully acknowledge the valuable support of Tomas Harryson along with the Innovation & Technology and Business Finance teams.

Gustaf Asplund, Felix Bjerhem Aronsson, Gothenburg, 2026-09-24

Declaration of AI Usage

In the development of this project and the accompanying thesis, Artificial Intelligence (AI) tools were employed solely in an assistive capacity, in areas such as coding and writing. In the context of coding, AI was used to accelerate certain aspects of development, to aid in debugging, and to support the exploration of methodological considerations. In the context of writing, AI was used for refinement purposes, including the identification of spelling and grammatical errors, the suggestion of alternative phrasings to improve flow, and for preliminary critique of the content. It was further used to help maintain a consistent academic tone throughout the thesis.

In all cases, AI served only to support and accelerate the authors' own work. It did not make any decisions, nor did it independently produce any part of the thesis. All substantive content, analysis, and judgments are the authors' own. The authors take full responsibility for any and all content presented in this work. All AI-assisted output was reviewed rigorously and verified by the authors, and any shortcomings or errors remain the responsibility of the authors alone.

Any sensitive use of AI was conducted exclusively through existing company tooling and agreements, ensuring that no confidential or proprietary information was disclosed to unauthorized providers. These agreements guarantee that submitted data is kept confidential and is not retained or used to train external models. General, non-sensitive assistance may have made use of other tools, but no sensitive material was shared outside of approved channels.

Gustaf Asplund, Felix Bjerhem Aronsson, Gothenburg, 2026-09-24

Contents

List of Figures xv

List of Tables xvii

List of Acronyms xix

Glossary xxi

1 Introduction 1
- 1.1 Work Aim . . . 2
- 1.2 Problem Formulation . . . 2
- 1.3 Research Questions . . . 3
- 1.4 Limitations . . . 4
  - 1.4.1 Hardware and Infrastructure . . . 4
  - 1.4.2 Architectural Scope . . . 4
  - 1.4.3 Data Generalizability . . . 4
  - 1.4.4 Cloud Environment Noise . . . 5
- 1.5 Risk Analysis and Ethical Considerations . . . 5
  - 1.5.1 Cloud Infrastructure Variability . . . 5
  - 1.5.2 Financial and Resource Constraints . . . 5
  - 1.5.3 Workload Representativeness . . . 6
  - 1.5.4 Sustainable Computing . . . 6
  - 1.5.5 Data Governance and Integrity . . . 6

2 Technical Background 7
- 2.1 Machine Learning . . . 7
  - 2.1.1 Neural Network . . . 7
    - 2.1.1.1 N-BEATS and N-HiTS . . . 9
    - 2.1.1.2 TSMixer . . . 9
  - 2.1.2 LSTM . . . 9
    - 2.1.2.1 xLSTMTime . . . 12
  - 2.1.3 Transformers and attention . . . 12
    - 2.1.3.1 Temporal Fusion Transformer . . . 13
  - 2.1.4 Convolutional Neural Network . . . 14
    - 2.1.4.1 TimesNet . . . 15
  - 2.1.5 Mixture of Experts . . . 15

2.1.5.1 Moirai-MoE . . . 16
2.1.6 Graph Neural Network . . . 16
2.1.6.1 TimeFilter . . . 16
2.1.7 Variational Autoencoders . . . 17
2.1.7.1 K^2VAE . . . 18
2.2 Hardware and Performance . . . 18
2.2.1 FLOPS . . . 18
2.2.2 Memory Bandwidth . . . 19
2.2.3 Arithmetic Intensity . . . 20
2.2.4 The Roofline Model . . . 20
2.2.5 Latency and Throughput . . . 21
2.2.6 Pareto Analysis . . . 22
2.3 Performance Profiling . . . 23
2.3.1 Hardware Profiling . . . 23
2.3.2 Framework-Level Profiling . . . 24
2.3.3 Empirical Roofline Toolkit . . . 25
2.4 Slurm and Environment Modules . . . 25

3 Methods **27**
3.1 Literature Review and Selection . . . 27
3.1.1 Model Selection . . . 27
3.1.2 Hardware Selection . . . 29
3.2 Dataset . . . 29
3.2.1 Dataset Characteristics . . . 29
3.2.2 Data Preprocessing . . . 30
3.3 Forecasting Task . . . 32
3.4 Benchmarking Pipeline . . . 32
3.4.1 Benchmarking Environment . . . 33
3.4.2 Model Harness and Porting . . . 35
3.4.3 Multivariate Adaptation and Batching Strategy . . . 36
3.5 Pareto and Roofline Analysis . . . 38
3.5.1 Cost Calculations . . . 38
3.5.2 Pareto Dimensions . . . 38
3.5.3 Roofline . . . 39

4 Results **41**
4.1 Model Choice . . . 41
4.1.1 Models Excluded Due to Runtime . . . 41
4.2 Hardware Choice . . . 42
4.3 Baseline Performance . . . 42
4.4 Hardware Resource Utilization . . . 44
4.5 Real-World Divergence . . . 47
4.6 Scaling and Compute . . . 47
4.7 Cost-Efficiency . . . 50

5 Discussion **55**
5.1 Exclusion of Models . . . 55

5.2 Rooflines and Resource Utilization . . . 55
5.2.1 ERT and GPUs . . . 55
5.2.2 Roofline Bounds and Overall Utilization . . . 57
5.2.3 Individual Model Behavior . . . 58
5.3 Scaling . . . 58
5.3.1 General Scaling Behavior . . . 59
5.3.2 Architectural Behaviors . . . 59
5.3.3 Implications for Hardware Selection . . . 60
5.4 Cost-Latency Efficiency . . . 60
5.5 Synthetic and Real-World Data . . . 61

6 Conclusion **63**
6.1 Future Work . . . 64

Bibliography **65**

A Model Choices **I**
A.1 All Models Considered . . . I
A.2 Open-Source Models Scored and Chosen . . . II

B Hyperparameter Performance Characteristics **III**
B.1 K2VAE . . . III
B.2 N-HiTS . . . IV
B.3 TFT . . . V
B.4 TimeFilter . . . VI
B.5 TSMixer . . . VII
B.6 xLSTMTime . . . VIII

C Per-Model Cost-Latency Pareto Frontiers **IX**
C.1 K2VAE . . . IX
C.2 N-HiTS . . . IX
C.3 TFT . . . X
C.4 TimeFilter . . . X
C.5 TSMixer . . . X
C.6 xLSTMTime . . . XI

D Per-Machine Rooflines **XIII**
D.1 Empirical Rooflines . . . XIII
D.1.1 F8als_v6 . . . XIII
D.1.2 F8s_v2 . . . XIV
D.1.3 F16als_v6 . . . XIV
D.1.4 F16s_v2 . . . XV
D.1.5 F32als_v6 . . . XV
D.1.6 F32s_v2 . . . XVI
D.2 Theoretical GPU Rooflines . . . XVI
D.2.1 NC8as_T4_v3 . . . XVI
D.2.2 NC16as_T4_v3 . . . XVII

D.2.3 NC40ads_H100_v4 XVII
D.2.4 NC64as_T4_v3 XVIII
D.2.5 NC80adis_H100_v4 XVIII

List of Figures

2.1 Mathematical model of an artificial neuron, illustrating inputs, weights, bias, and the activation function. 8

2.2 Architecture of a standard Multilayer Perceptron with two hidden layers, where each node in a given layer is fully connected to every node in the next. 8

2.3 High-level comparison of network architectures. **(a)** A standard ANN processes data purely feed-forward. **(b)** An RNN introduces a recurrent loop, passing the hidden state h_{t-1} back into the network for the current time step t. 10

2.4 Unrolled architectures through time. **(a)** A standard RNN passes only a single hidden state (h_t) between time steps, which is susceptible to the vanishing gradient problem. **(b)** An LSTM introduces a parallel track for the cell state (c_t), acting as a long-term memory highway that mitigates gradient degradation. 11

2.5 Computing a 2D convolution by sliding the kernel K across the input I. Three example kernel positions are highlighted in different colors; each produces the correspondingly coloured entry of the feature map S. The same kernel weights are reused at every valid position. 14

2.6 A simple roofline example, showing the compute against a single memory. 21

2.7 Simple roofline extended with a ridge point marker 21

2.8 Roofline with multiple bandwidth slopes for different tiers of memory 22

2.9 An illustrative Pareto frontier for a hypothetical problem with two competing objectives, cost and accuracy. Models A–E are Pareto-optimal and trace out the frontier, while models F, G, and H are dominated, at least one frontier model achieves both lower cost and higher accuracy. 23

3.1 A qualitative representation of the underlying dataset characteristics. The diagram illustrates the long-running multi-year trend alongside substantial intra-year variance caused by continuous agreements. Most notably, it highlights the sharp, renewal-driven seasonality that concentrates at the start of each yearly period. 30

3.2 Comparison of execution patterns between natively multivariate architectures and adapted univariate models. Native models (A) process the full panel jointly, sharing state and raising arithmetic intensity. Adapted models (B) require slicing the input into independent sequences and relying on the batch dimension, which isolates the model state and shifts the hardware profile toward memory traffic. 37

4.1 Comparison between models and SKUs for roofline 44
4.2 Model collection roofline for the F8s_v2 node. 45
4.3 Model collection roofline for the NC40ads_H100_v4 node. 45
4.4 Model collection roofline for the theoretical roofline of the NC40ads_H100_v4 node. 46
4.5 Real data vs. synthetic data divergence between SKUs 47
4.6 Real data vs. synthetic data divergence between models 48
4.7 Cost-latency across models, p50 50
4.8 Cost-latency across models, p99 51
4.9 Cost-Latency for TSMixer, p50 52
4.10 Cost-Latency for TSMixer, p99 52
4.11 Cost-Latency for N-HiTS, p50 53
4.12 Cost-Latency for N-HiTS, p99 53

List of Tables

4.1 Selected models and their corresponding classes. 41
4.2 Chosen hardware instances. NVIDIA GPUs are assumed. Prices retrieved as of 2026-05-05. 42
4.3 Achieved empirical hardware roofline performance 43
4.4 Theoretical hardware roofline performance 43
4.5 Characteristics across trials for TSMixer 49
4.6 Characteristics across trials for TimeFilter 49

List of Acronyms

K^2VAE	Koopman-Kalman Enhanced Variational Autoencoder
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ARIMA	Auto-Regressive Integrated Moving Average
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DRAM	Dynamic Random-Access Memory
ERP	Enterprise Resource Planning
ERT	Empirical Roofline Toolkit
FFT	Fast Fourier Transform
FLOPS	Floating-Point Operations Per Second
FPGA	Field-Programmable Gate Array
GDPR	General Data Protection Regulation
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRN	Gated Residual Network
LLM	Large Language Model
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multi-Layer Perceptron
mLSTM	Matrix Long Short-Term Memory
MoE	Mixture of Experts
N-BEATS	Neural Basis Expansion Analysis for Time Series
N-HiTS	Neural Hierarchical Interpolation for Time Series
NLP	Natural Language Processing
OI	Operational Intensity
PMC	Performance Monitoring Counter
RNN	Recurrent Neural Network
sLSTM	Scalar Long Short-Term Memory
TFT	Temporal Fusion Transformer
TPU	Tensor Processing Unit
TSMixer	Time-Series Mixer

VAE	Variational Autoencoder
VM	Virtual Machine
xLSTM	Extended Long Short-Term Memory

Glossary

Foundational model	An AI model pre-trained on large and diverse datasets that can be used for adapting to more specific models.
SKU	A unique identifier for each distinct product and service that can be purchased
YAML	A structured data serialization language.

1

Introduction

In the domain of enterprise financial planning, organizations are increasingly shifting from high-level, monthly statistical forecasting (e.g. Auto-Regressive Integrated Moving Average [ARIMA]) to granular Artificial Intelligence (AI)-driven approaches utilizing deep learning or gradient boosting [1]. While these advanced models strive to capture non-linear dependencies and achieve actionable insights at the product level, they introduce a massive increase in computational complexity. Unlike legacy batch processes, modern forecasting pipelines must balance prediction accuracy with strict operational constraints regarding latency, throughput, and cloud infrastructure costs [2].

This transition presents an important infrastructure challenge. Compute options range from local Central Processing Units (CPUs) to cloud Graphics Processing Units (GPUs) and auto-scaling clusters, differing drastically in performance profiles and cost structures. A significant disconnect often exists between data science teams, who optimize for algorithmic accuracy, and platform engineers, who must ensure operational efficiency. Without a systematic understanding of how specific forecasting workloads scale, companies often select hardware based on heuristics. This lack of insight is significant because the relationship between hardware architecture (e.g. memory bandwidth, parallelization capabilities) and model inference behavior is often non-linear [3].

Furthermore, the economic profile of machine learning in production is shifting. While training is a one-time cost, inference is more continuous and accumulates with every prediction request. In high-throughput scenarios, such as generating daily forecasts for millions of Stock Keeping Units (SKUs), inefficient resource utilization can lead to substantial financial waste. For example, deploying a memory-bound recurrent model on a compute-dense GPU often results in "starved" compute units, where the accelerator remains underutilized while the organization pays for peak performance capabilities [4].

Existing research has attempted to address these inefficiencies through dynamic serving frameworks. Systems like INFaaS [5] and Perseus [6] demonstrate that intelligent hardware selection can significantly reduce serving costs. However, these frameworks primarily focus on computer vision or natural language processing. This creates a blind spot for forecasting workloads. While modern forecasting architectures often share a Transformer-based backbone with Large Language Models (LLMs), they frequently operate on much longer input horizons. This activates the quadratic

computational complexity with respect to sequence length, which is a fundamental characteristic of self-attention [7], creating unique scaling profiles that generic benchmarking can fail to capture.

Therefore, a clear gap remains in understanding the trade-off between model complexity, latency requirements, and runtime cost specifically for time-series data. Could a specific hardware configuration allow for a more complex model without exceeding the budget? Conversely, can a lower-cost setup meet the latency targets of a real-time system? This thesis proposes a workload-aware benchmarking framework to bridge this gap, transitioning infrastructure decisions from "black-box" heuristics to data-driven provisioning.

1.1 Work Aim

The primary aim of this thesis is to empirically characterize the computational resource consumption of diverse deep learning forecasting architectures. The objective is to develop a cross-architectural benchmarking framework that quantifies the trade-offs between algorithmic complexity, hardware utilization, and operational cost.

We also aim to ensure the industrial transferability of the benchmarking results. Synthetic benchmarks can lead to results that diverge from production reality, leaving engineers with potentially unreliable data for infrastructure planning. By anchoring this study in a real-world context, we aim to capture the specific performance behaviors that can only be found in real-world applications. This ensures the framework delivers actionable insights, allowing system architects to trust that the measured cost-latency trade-offs will hold true in actual deployment scenarios.

Specifically, this work aims to:

- Characterize computational profiles through implementation and benchmarking a various state-of-the-art Deep Learning (DL) forecasting architectures representing distinct execution patterns.
- Analyze how these architectures differ in their Operational Intensity (OI). We aim to map where each architecture falls on the theoretical roofline model of the target hardware.
- Synthesize the results to identify which combinations of architecture and hardware selections lie on the Pareto frontier for cost, latency and throughput.

Ultimately, this work aims to provide a framework for system architects, allowing them to select the most cost-efficient hardware-model pair based on their specific latency constraints and data granularity.

1.2 Problem Formulation

The shift from statistical forecasting to DL has introduced significant architectural heterogeneity into financial planning pipelines [8]. Unlike ARIMA models, which

are largely CPU-bound and scalar, modern DL architectures exhibit diverse computational profiles that interact inefficiently with general-purpose cloud hardware.

The core systems challenge is the variance in OI. Different forecasting architectures stress hardware distinctively:

- Recurrent architectures (e.g. Long Short-Term Memory [LSTM]) often exhibit low OI due to sequential dependencies that inhibit parallelization [9], leading to memory-latency bound execution.
- Attention-based architectures (e.g. transformers) exhibit quadratic complexity $\mathcal{O}(L^2)$ and higher parallelization potential [7], but risk becoming memory-bandwidth bound during inference on sparse, high-granularity data [10].

Due to the nature of synthetic data generation models, some forms of biases may be generated that do not describe the sparsity and irregularity of real-world Enterprise Resource Planning (ERP) data [11]. This could lead to "black box" deployment scenarios where system architects cannot predict if a workload will saturate the compute units (Floating-Point Operations Per Second (FLOPS)) or stall on memory bandwidth. As future prediction accuracy is crucial, real-world data is compared to synthetic data to infer transferability of results, showing the potential gap between them for real-world deployments.

1.3 Research Questions

Formally, this creates a multi-objective optimization problem. The goal is to identify a tuple of $(Architecture, Hardware)$ that minimizes cost while strictly adhering to production latency constraints:

$$\min_{a,h} \quad C(a,h) \tag{1.1}$$

$$\text{subject to} \quad \text{Latency}_{p99}(a,h) \leq \mathcal{L}_{max} \tag{1.2}$$

$$\text{and} \quad \text{Throughput}(a,h) \geq \mathcal{T}_{target} \tag{1.3}$$

Where $a \in \mathcal{A}$ is the set of model architectures and $h \in \mathcal{H}$ is the set of available hardware SKUs. To solve this, we must characterize the Pareto Frontier of these trade-offs. This thesis addresses the following research questions:

- RQ1 - Roofline characterization: How does the OI of distinct forecasting architectures (e.g. Recurrent Neural Networks (RNNs) vs. transformer) compare against the roofline limitations of commodity cloud instances?
- RQ2 - Real-world divergence: To what extent does the cost-latency trade-off diverge when benchmarking against real-world, irregular ERP data compared to the theoretical performance implied by simpler synthetic data?

- RQ3 - Pareto optimization: How can a cross-architectural benchmarking framework identify a global Pareto frontier that allows system architects to select the optimal hardware-model pair for specific latency constraints?

1.4 Limitations

To ensure the feasibility of the project within the master's thesis timeframe, the following boundaries have been established regarding the scope of hardware, model architectures, and generalizability.

1.4.1 Hardware and Infrastructure

While the benchmarking framework aims to be distinct from specific hardware vendors, the experimental evaluation is limited to the infrastructure available through the partner company's cloud provider, Microsoft Azure. Therefore the study focuses exclusively on x86-64 CPUs and available GPUs.

Specialized hardware accelerators such as Tensor Processing Units (TPUs), Field-Programmable Gate Arrays (FPGAs), or edge-inference devices (e.g. NVIDIA Jetson) are explicitly excluded from the scope. Furthermore, we did not account for "cold-start" latencies associated with serverless functions, focusing instead on the steady-state inference performance of provisioned instances.

Additionally, spot and reserved-type instances were not tested. This is due to our harness needing to run uninterrupted, and metrics likely transferring purely based on hourly pricing, as hardware does not change.

1.4.2 Architectural Scope

The thesis focuses on the systems aspect of serving forecasting models rather than the data science aspect of inventing new model architectures. Therefore, we did not try to develop completely new and novel forecasting algorithms. The study is limited to benchmarking a representative subset of existing state-of-the-art DL-based architectures (e.g. Temporal Fusion Transformers, LSTM-based xLSTM, or N-HiTS) implemented in standard frameworks, such as PyTorch. Low-level compiler optimizations (e.g. writing custom CUDA kernels or operator fusion beyond what standard compilers like TorchScript/ONNX Runtime provide) are outside the scope.

1.4.3 Data Generalizability

The benchmarking results were derived from the specific historical ERP data provided by the partner company. This data is characterized by specific sparsity patterns, seasonality, and stationarity inherent to enterprise financial planning. While the methodology for benchmarking is intended to be generalizable, the specific quantitative conclusions regarding cost-latency trade-offs may not directly transfer to domains with drastically different signal characteristics (e.g. high-frequency audio processing or continuous weather sensing).

1.4.4 Cloud Environment Noise

As experiments occur in a cloud environment, perfect isolation of hardware resources cannot be guaranteed. Variance caused by multi-tenancy interference or hypervisor scheduling is an inherent limitation of cloud benchmarking. While we employ statistical methods (e.g. repeated runs, removing outliers) to mitigate this, strictly deterministic performance measurements are not possible without bare-metal access.

1.5 Risk Analysis and Ethical Considerations

This thesis operates at the intersection of commercial cloud systems and academic research. As such, it faces specific operational challenges regarding reproducibility and resource constraints, alongside ethical obligations regarding computational efficiency and data handling. The following sections outline the potential risks and the strategies adopted to address them, as well as the broader ethical implications of the work.

1.5.1 Cloud Infrastructure Variability

The primary technical risk to this study is the inherent non-determinism of public cloud infrastructure. Benchmarking on virtualized platforms introduces performance jitter caused by hypervisor scheduling and multi-tenant resource contention. This makes strictly deterministic measurements impossible and threatens the reproducibility of the results.

To address this, the experimental methodology moves beyond simple timing measurements to employ a more robust statistical analysis. Rather than relying on single-shot inference runs, the benchmark executes repeated warm-up and measurement loops to establish a confidence interval for each hardware configuration.

1.5.2 Financial and Resource Constraints

The reliance on high-performance accelerators introduces a significant financial risk. High-end GPU instances incur substantial hourly costs, and a search over deep learning hyperparameters could rapidly deplete the thesis budget before meaningful data is collected.

Consequently, the project adopts a tiered development strategy. The development and debugging of the benchmarking pipeline was conducted exclusively on local hardware or low-cost instances. The high-cost instances were strictly reserved for the final execution of the validated benchmark suite, ensuring that the budget is allocated to the generation of final results.

1.5.3 Workload Representativeness

A scientific risk exists regarding the density of the proprietary dataset provided. If the historical ERP data is not sufficiently large to saturate the memory bandwidth of high-end instances, the benchmarks may result in trivial findings where the hardware is consistently underutilized.

To mitigate this, the benchmarking framework is designed to treat dataset size as a variable rather than a constant. If the historical data prove insufficient to stress the hardware, the workload could be synthetically augmented, by duplicating sequences or artificially increasing batch sizes, to ensure the system can identify the saturation point and the "break-even" threshold between CPU and GPU performance.

1.5.4 Sustainable Computing

From an environmental perspective, this thesis directly addresses the principles of sustainable computing. The training and serving of deep learning models contribute to a growing global carbon footprint. By replacing static, heuristic-based provisioning with a workload-aware benchmarking framework, this project aims to reduce the "over-provisioning" of infrastructure. Identifying the precise hardware requirements for a given latency target minimizes the deployment of oversized accelerators, thereby directly reducing the energy waste associated with idle compute resources.

1.5.5 Data Governance and Integrity

While the project utilizes proprietary financial data, it does not directly use or access any sensitive personal data subject to General Data Protection Regulation (GDPR). The dataset instead contains business metrics, which are commercially sensitive. Therefore, this publication presents only a subset of anonymized data, ensuring that no strategic business information is exposed. Additionally, all work involving the data did not leave the company's property, which lowered the chance of accidental leaks.

2

Technical Background

In this chapter, we discuss relevant technologies that are mentioned in this report.

2.1 Machine Learning

Defined as a subset of Artificial Intelligence, Machine Learning (ML) allows machines to learn patterns from data to perform tasks that can otherwise not be programmed [12]. This process involves training models on datasets to generalize to unseen data. ML is typically divided into three main paradigms.

Supervised learning involves training on labeled data to perform tasks such as classification or regression [13]. Unsupervised learning deals with unlabeled data, aiming to infer the underlying probability density or structure of the data, such as through clustering or trend analysis [14]. Finally, reinforcement learning focuses on how the algorithm should take actions in an environment in order to maximize the notion of a reward. Unlike supervised learning, the agent is not told which actions to take, but instead must discover which produce the most reward by trying them [15].

Time-series prediction is generally approached as a supervised learning task [16]. The model is trained using historical data sequences to predict subsequent values [17]. Through this process, the model learns the underlying mechanisms and temporal behaviors of the data, enabling it to make accurate forecasts based on current trends, such as ERP information as explored in this thesis.

2.1.1 Neural Network

Inspired by the biological structure of the brain, an Artificial Neural Network (ANN) is an ML model that processes data through interconnected units known as artificial neurons or nodes. Each node receives a collection of inputs, computes a weighted sum along with an added bias term, and passes the result through an often non-linear activation function to determine its output. [18] Mathematically, the output y of a single artificial neuron can be expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f\left(\mathbf{w}^\top \mathbf{x} + b\right), \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input vector, $\mathbf{w} \in \mathbb{R}^n$ the corresponding weight vector, $b \in \mathbb{R}$ the bias term, and $f(\cdot)$ a (typically non-linear) activation function. When m such

neurons are stacked into a single layer, the per-neuron weights become rows of a weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ and the biases form a vector $\mathbf{b} \in \mathbb{R}^m$, giving the compact layer-wise form

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{2.2}$$

where f is applied element-wise. This mathematical model can be visualized in Figure 2.1.

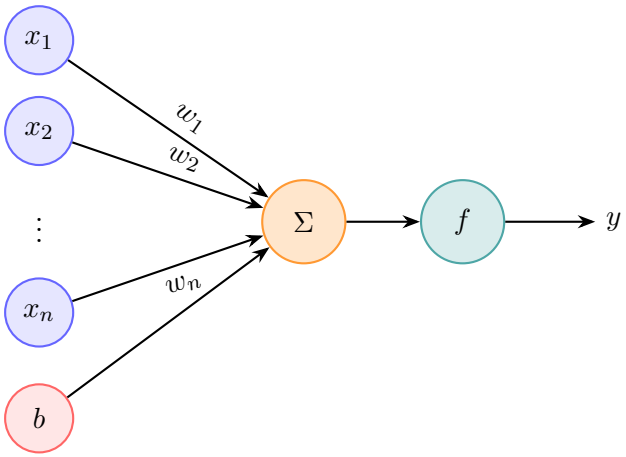


Figure 2.1: Mathematical model of an artificial neuron, illustrating inputs, weights, bias, and the activation function.

When these nodes are organized into distinct, sequential layers, specifically an input layer, one or more hidden layers, and an output layer, the architecture is commonly referred to as a Multi-Layer Perceptron (MLP) [19]. This is best visualized in Figure 2.2.

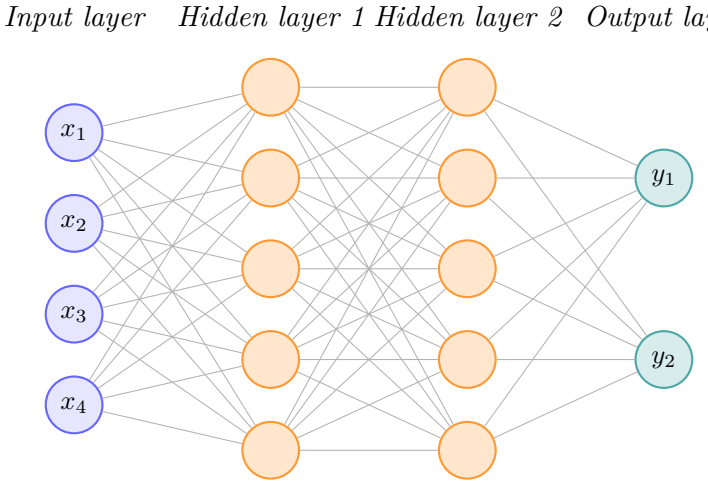


Figure 2.2: Architecture of a standard Multilayer Perceptron with two hidden layers, where each node in a given layer is fully connected to every node in the next.

2.1.1.1 N-BEATS and N-HiTS

The Neural Basis Expansion Analysis for Time Series (N-BEATS) [20] introduced a pure deep learning architecture tailored for interpretable forecasting. Unlike typical recurrent or convolutional networks, N-BEATS relies on a stack of fully connected layers arranged in a "doubly residual" topology. Each block in the stack performs two operations: it *backcasts* (reconstructs) the input signal to remove the signal explained by the current block, and it *forecasts* partial predictions which are summed globally. In its interpretable configuration, these blocks project the signal onto specific basis functions (e.g., polynomials for trend, Fourier series for seasonality).

Building upon this foundation, Neural Hierarchical Interpolation for Time Series (N-HiTS) [21] was proposed to address the computational volatility of long-horizon forecasting. N-HiTS incorporates a hierarchical data sampling strategy (multi-rate pooling) that allows different stacks to specialize in different frequency bands. By replacing the local constant projections of N-BEATS with "local nonlinear projections onto basis functions across multiple blocks" via hierarchical interpolation [21], N-HiTS significantly reduces the parameter count and improves predictive accuracy over long horizons.

2.1.1.2 TSMixer

Time-Series Mixer (TSMixer) [22] another all-MLP based architecture for multivariate time series forecasting. Its central building block is the mixer layer, which consists of two residual MLP blocks applied in alternation. The first of these blocks is a time-mixing MLP acting along the temporal dimension and shared across all features. The second is a feature-mixing MLP acting along the feature dimension and shared across all time steps. The time-mixing block is responsible for capturing temporal patterns, while the feature-mixing block tries to capture cross-variate dependencies. This weight sharing keeps parameter growth at $\mathcal{O}(L + C)$ in the lookback length L and the number of variates C, rather than the $\mathcal{O}(LC)$ that a fully-connected MLP would have.

A full forecast is produced by stacking N such mixer layers, each wrapped in residual connections and 2D normalisation across both the time and feature dimensions, followed by a linear temporal projection that maps the representation from the lookback length L to the forecast horizon T.

2.1.2 LSTM

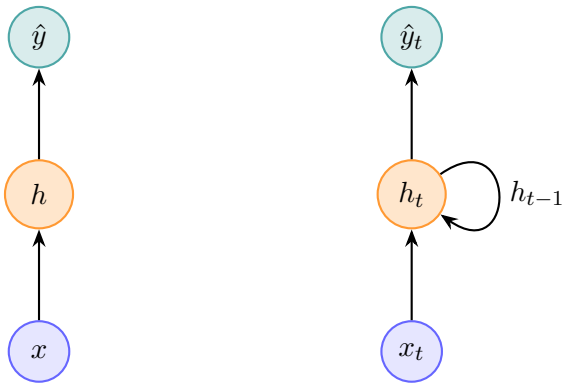
The foundational architecture behind LSTMs lies in RNNs. While a standard feed-forward ANN computes its hidden representations based solely on the current input,

$$h = \sigma(Wx + b) \tag{2.3}$$

an RNN introduces a recurrent connection to maintain an internal hidden state [23]. Specifically, the hidden state at time step t is computed using both the current input x_t and the hidden state from the previous time step h_{t-1}:

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h) \tag{2.4}$$

This recursive calculation carries information across sequential time steps, allowing the network to leverage past context for current predictions, as illustrated in Figure 2.3. However, standard RNNs struggle to capture true long-term dependencies due to the vanishing gradient problem [24]. During backpropagation, the error gradients calculated at the end of a sequence diminish exponentially as they are repeatedly multiplied by the recurrent weight matrix while propagating backward, effectively causing the network to "forget" the earliest inputs.



(a) Standard ANN (b) Standard RNN

Figure 2.3: High-level comparison of network architectures. **(a)** A standard ANN processes data purely feed-forward. **(b)** An RNN introduces a recurrent loop, passing the hidden state h_{t-1} back into the network for the current time step t.

LSTMs [25], [26] solve this by introducing a dedicated cell state, c_t, to act as a long-term memory. The structural advantage of this addition is visualized in Figure 2.4, which illustrates how the LSTM establishes a parallel memory highway for c_t to bypass the standard recurrent bottleneck. This is achieved by adding a series of learned "logic gates" where the network can choose to update and reset this internal state based on the current input, x_t, and previous hidden state, h_{t-1}. Specifically, a forget gate (f_t) [27] and an input gate (i_t) dictate what information is discarded or stored:

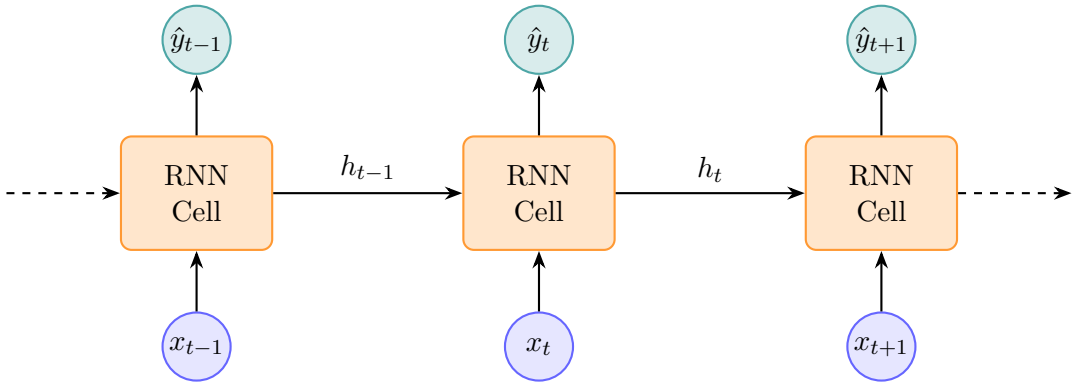
$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + p_f \odot c_{t-1} + b_f) \tag{2.5}$$

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + p_i \odot c_{t-1} + b_i) \tag{2.6}$$

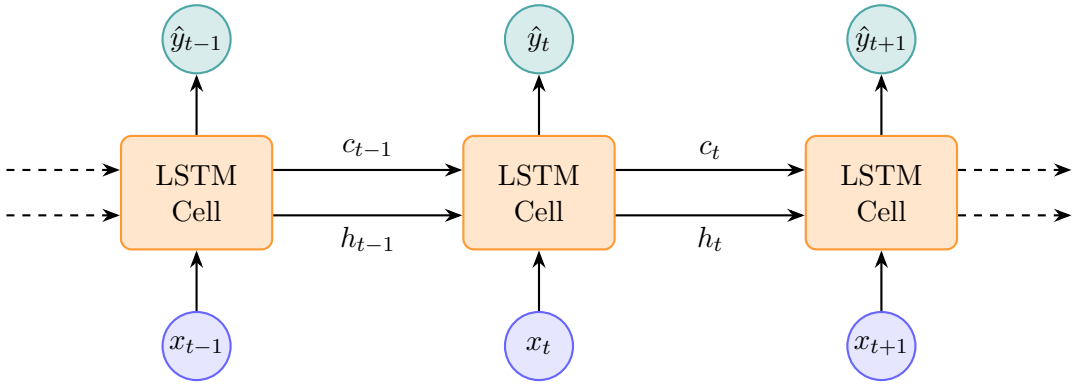
This gating mechanism lets the network learn to ignore saving noisy values from the data and only keep the most relevant information for future predictions, which is then formalized in the updated cell state:

$$c_t = z_t \odot i_t + c_{t-1} \odot f_t \tag{2.7}$$

Finally, an output gate determines what part of this long-term memory becomes the new short-term hidden state (h_t). By actively managing this information flow, the architecture successfully moves the capabilities of the RNN from a short-term horizon to a more long-term one.



(a) Unrolled RNN



(b) Unrolled LSTM

Figure 2.4: Unrolled architectures through time. **(a)** A standard RNN passes only a single hidden state (h_t) between time steps, which is susceptible to the vanishing gradient problem. **(b)** An LSTM introduces a parallel track for the cell state (c_t), acting as a long-term memory highway that mitigates gradient degradation.

2.1.2.1 xLSTMTime

While the traditional LSTM-based systems described above can capture long-term dependencies better than traditional RNNs, several weaknesses remain that motivate further improvements. The first concerns the gate activations. Rather than employing sigmoid-activated gates to govern when to override the stored state, Extended Long Short-Term Memory (xLSTM) uses an exponential activation function. This alleviates the issue that the network can struggle to revise its previous storage decisions once they have been made, the unbounded nature of the exponential allows the network to *strongly* override prior storage decisions when needed. As a result, the model adapts more readily to new inputs and, most importantly, more easily updates its internal state when more relevant information is ingested [28].

The second important improvement of xLSTM over LSTM concerns memory. The xLSTM architecture introduces two new types of memory blocks, Scalar Long Short-Term Memory (sLSTM) and Matrix Long Short-Term Memory (mLSTM). The former retains the traditional scalar memory cell but introduces a new form of memory mixing where multiple cells are grouped into heads, allowing interactions to occur between cells within the same head but not across heads. The latter draws inspiration from transformers and instead employs a matrix-valued memory $C_t \in \mathbb{R}^{d \times d}$, which is updated using a covariance (outer-product) rule:

$$C_t = f_t \, C_{t-1} + i_t \, v_t k_t^\top \tag{2.8}$$

Storage is performed by accumulating outer products of key-value pairs into C_t, while retrieval is performed via matrix multiplication, $\tilde{h}_t = C_t q_t$, which is then normalized using a corresponding normalizer state. This formulation considerably increases storage capacity and, by removing the hidden-to-hidden recurrent connection entirely, enables the forward pass to be computed in parallel, making it well-suited to GPUs.

These two block types then form the xLSTM architecture through residual stacking, where the ratio of sLSTM to mLSTM blocks can vary.

2.1.3 Transformers and attention

While LSTMs successfully extend the memory horizon of RNNs, they remain inherently sequential. The computation of the hidden state h_t strictly relies on h_{t-1}, creating a computational bottleneck that limits parallel processing. The transformer architecture [7] solves this by removing recurrence entirely and relying instead on a "self-attention" mechanism.

Rather than maintaining a hidden state over time, self-attention evaluates entire sequences simultaneously. It achieves this by projecting the input sequence into three learned matrices: Queries (Q), Keys (K), and Values (V).

The model computes attention scores to determine how much focus each element in the sequence should place on every other element. This is done by taking the dot product of the queries and keys. To prevent these values from growing too large

and causing vanishing gradients in the subsequent softmax function, the scores are scaled by the inverse square root of the key dimension, d_k:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.9}$$

In this equation, $\frac{QK^T}{\sqrt{d_k}}$ generates a matrix representing the contextual similarities across the sequence. The softmax function then converts these into normalized weights, which dictate how much of each corresponding value in V is expressed in the output.

Because this mechanism relies on matrix multiplications instead of sequential steps, transformers can efficiently learn complex, non-linear patterns and capture true long-range dependencies in parallel. This architecture has become highly influential in Natural Language Processing (NLP) and has naturally extended into other disciplines, including time-series forecasting.

2.1.3.1 Temporal Fusion Transformer

By treating discrete time-steps as tokens, a transformer model can be used to predict time series. The Temporal Fusion Transformer (TFT) is an example of such an implementation [29]. The focus of the architecture is multi-horizon forecasting, which involves 3 different input types, namely static covariates, the past datapoints, and sometimes known future inputs (e.g. holidays etc. that could be relevant to the domain). Previous models handled this poorly, either assuming all inputs are known into the future, or flattening the metadata into the time-varying stream. Instead, TFT is a direct multi-horizon forecaster, which predicts all horizons simultaneously and outputs multiple quantiles at each horizon to form prediction intervals.

In terms of implementation, this is handled by letting each type of input pass through its own variable selection network, where weights are produced over features at each step. Static covariates also go through an additional step of dedicated encoders that create context vectors. These vectors are then injected into the variable selection, the integrated LSTM for initial states, and then into a static enrichment layer. This makes the static metadata condition the temporal processing throughout. These components are built around Gated Residual Networks (GRNs), which let the model skip non-linear processing where it isn't needed, enhancing efficiency.

The previously mentioned LSTM is created for sequence-to-sequence handling, encoding the past inputs and decoding the future known inputs, handling local patterns and replacing positional encoding. Then, a multi-head self-attention layer focuses on the long-range dependencies, which is slightly modified from the standard mechanism by sharing values across heads so attention weights also remain interpretable. A linear projection on top of the decoder output then produces the final quantile forecasts (e.g. P10/P50/P90), trained with quantile loss.

2.1.4 Convolutional Neural Network

Originally taking inspiration from neuroscientific principles [30], a Convolutional Neural Network (CNN) is a well-established and high-performing architecture for completing tasks such as image recognition [31]. CNNs in essence work by repeatedly applying a mathematical operation called convolution. Goodfellow et al. [32] describe the following operations and properties of CNNs. For a 2D input I and kernel K, a discrete convolution is defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)\, K(i-m, j-n). \tag{2.10}$$

Each output element $S(i,j)$ is therefore a weighted combination of input values in a local neighborhood around (i,j), with the kernel K supplying the weights. The full output array is referred to as a feature map. A more visual representation of this can be seen in Figure 2.5.

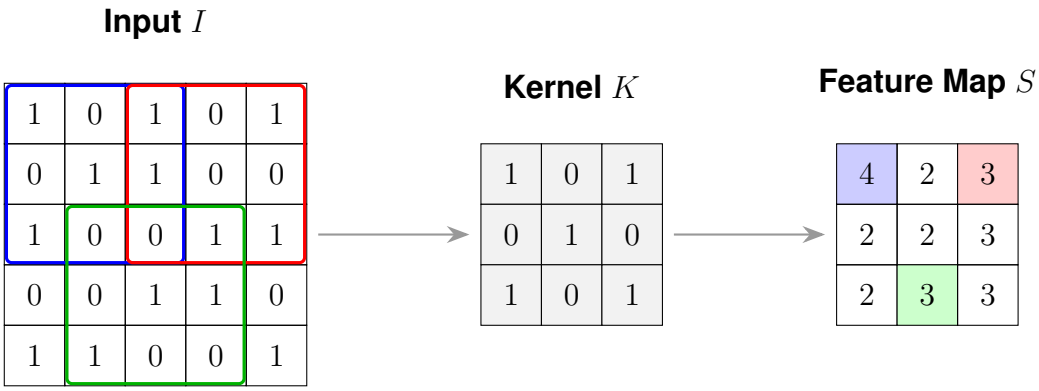


Figure 2.5: Computing a 2D convolution by sliding the kernel K across the input I. Three example kernel positions are highlighted in different colors; each produces the correspondingly coloured entry of the feature map S. The same kernel weights are reused at every valid position.

These convolutions become various kinds of feature extractors, which could extract indicators of, for example, straight lines, curves, edges and other shapes (or patches of colors). The kernels themselves are learned as part of the network, meaning the network itself can learn which features it deems important, and more accurately pass through the most important information to later layers. This gives rise to a hierarchy of features, where early layers tend to learn low-level patterns like edges and colors, and deeper layers compose these into textures, parts, and eventually higher-level concepts.

A defining property of CNNs is that the same kernel is applied across the entire input, meaning a feature detector learned in one location is reused everywhere. Additionally, because the kernel is much smaller than the input, each output element depends on only a small local region of the input, which is known as sparse interactions, which keeps memory and compute requirements reasonable even for large

inputs. This makes CNNs parameter-efficient and gives them a built-in sensitivity to spatial structure.

Convolutional layers are also typically interleaved with nonlinear activation functions (such as ReLU), without which stacking convolutions would collapse to a single linear operation. Pooling layers are also commonly applied to downsample feature maps, which reduces computation and provides some degree of translation invariance. The output of these stacked operations is then passed to traditional ANN layers for learning the task at hand.

2.1.4.1 TimesNet

The basis on which the TimesNet architecture is built is the foundational thinking that in most time-series applications, there are both intraperiod and interperiod variations. These periods could, for example, handle the variations within a single day (intraperiod) as well as the fluctuations that happen from day to day over a year (interperiod). The main idea of TimesNet relies on mapping these 1-dimensional temporal variations into a 2-dimensional space, where the columns represent intraperiod variations and the rows represent interperiod variations [33].

These periods are identified using Fast Fourier Transform (FFT), which helps the model automatically pick the top-k most significant frequencies to determine the most appropriate period lengths. By identifying multiple periods, the model creates a set of different 2D tensors, which helps it model multiple overlapping periodicities (like years, months, and weeks) simultaneously.

This change in perspective allows the model to capture these variations by utilizing 2D convolutional kernels. This creates a CNN-based architecture that can efficiently recognize patterns in the time series similarly to how grouped pixels are recognized in an image. These patterns are processed using a TimesBlock, which contains a parameter-efficient Inception block used to aggregate variations at different granularities. Finally, these different periodic representations are transformed back to 1D and weighted based on the amplitudes of their original frequencies to create a unified final result.

2.1.5 Mixture of Experts

Scaling dense models to generalize across diverse feature sets comes with computational costs, as every parameter is activated for every input. The Mixture of Experts (MoE) paradigm addresses this via conditional computation. Instead of activating the entire network, a learnable gating mechanism dynamically routes inputs to a specific subset of "experts." This leads to only a fraction of the total parameters being active during any given forward pass, decoupling model capacity from inference cost.

Because the gating network and experts are optimized together, the model learns to cooperatively partition the problem space, allowing experts to specialize in distinct features [34]. However, this architecture presents deployment challenges. While computational cost is reduced, memory requirements remain high as all experts must

be loaded into VRAM. Furthermore, the discontinuous nature of the routing mechanism often makes MoE models more unstable and difficult to fine-tune compared to dense counterparts.

2.1.5.1 Moirai-MoE

The base model, Moirai [35], is a Transformer-based foundational model designed for universal time-series forecasting. To handle varying data frequencies, Moirai employs a multi-patch projection layer that embeds patches of different sizes into a unified latent space. The network is trained via a masked prediction objective, where it learns to reconstruct masked patches based on the context of observed values.

However, a monolithic dense model struggles to generalize across the vast heterogeneity of universal time-series data. As noted in [35], real-world series exhibit diverse distributions within short context windows, making it difficult for a single set of weights to capture all frequency patterns effectively. Moirai-MoE [36] mitigates this by integrating a MoE architecture. By using a gating mechanism to route inputs to specialized Transformer-based experts, the model can learn distinct patterns for different data distributions, improving both generalization and computational efficiency.

2.1.6 Graph Neural Network

Most deep learning architectures are constructed for processing data with a regular grid structure, such as images or sequential text. However, many real-world systems are naturally represented as complex, non-Euclidean structures consisting of interconnected entities. To address this, Graph Neural Network (GNN) architectures were developed, capable of learning directly from graph-based data.

The core mechanism behind these models relies on a process known as message passing. During this, each node in the graph iteratively aggregates information from its immediate neighbors to update its own latent representation [37]. This allows the model to capture both the structural dependencies of the graph and the individual node features simultaneously. These networks have since become a well-used approach for modeling complex relationships in tasks including node classification, link prediction, and molecular property forecasting.

2.1.6.1 TimeFilter

A more recent approach for applying GNNs to time series data is the TimeFilter model [38]. It constructs a graph by splitting the multivariate input into spatial-temporal patches, where each channel is divided into non-overlapping segments of fixed length, and every resulting segment is treated as an individual node. Given an input with C channels and N patches per channel, this yields a graph of $C \times N$ nodes.

In contrast to earlier GNN-based approaches for multivariate forecasting, which typically assign one node per variable and rely on a separate temporal module to capture

the sequential structure, this design unifies inter-variable and intra-variable dependencies within a single graph. Temporal relations become edges between patches of the same channel, spatial relations become edges between patches of different channels at the same time step, and spatial-temporal relations capture interactions between different channels across distinct time intervals.

Once this global, densely connected graph is established, TimeFilter decomposes it into patch-specific ego-graphs centered on each node. Each ego-graph is further partitioned by edge type, and a MoE gating mechanism is assigned per edge type as a filter that prunes the ego-graph prior to aggregation, so that each node retains only the subset of edges deemed relevant to its local context.

The filters are activated through a Top-p allocation scheme, in which experts are selected in descending order of confidence until their cumulative probability exceeds a given threshold. This allows the number of active dependency types to vary across nodes, aligning the dependency selection with the current context and firing only the experts relevant to a given prediction. Once filtering is complete, the ego-graphs are recombined and standard message passing is applied for forecasting. In this way, TimeFilter preserves the underlying aggregation mechanism of GNNs while augmenting it with a node-specific edge selection stage.

2.1.7 Variational Autoencoders

A classical autoencoder [32] learns by compressing the input as much as possible and then reconstructing it from that compressed representation. This works well for compression, but the bottleneck is a single fixed point in some latent space, not a distribution, so the model has no way of saying that a given input could plausibly correspond to multiple internal states. For forecasting, this is a real limitation since a fully deterministic model produces one prediction per input, with no means of expressing that several futures might be equally likely [39].

The Variational Autoencoder (VAE) [40], [41] addresses this by making the bottleneck probabilistic. Instead of mapping an input X to a single latent point, the encoder produces a Gaussian distribution $Q(z|X)$ over a latent vector z, described by a normal distribution. A point is then sampled from this distribution and passed to the decoder, which attempts to reconstruct X from it.

This means that training should balance two competing goals, the decoder should reconstruct the input as accurately as possible while the encoder's output distribution should stay close to a simple standard-normal distribution. This second goal keeps the latent space organized in a way where sampling any z should yield something the decoder could reasonably turn into a realistic output.

There's a problem with this approach though, which is that gradients cannot flow through a random sampling step. VAEs sidestep this with the reparameterization trick, drawing a sample from a fixed noise source $\epsilon \sim \mathcal{N}(0, I)$ and then shifting and scaling it using $\mu(X)$ and $\Sigma(X)$. The randomness now becomes part of the input, leaving a fully differentiable path through the network for learning.

VAEs are primarily focused as a kind of generative model, but various architectures exist that try to use them for prediction tasks, such as forecasting [42].

2.1.7.1 K^2VAE

The main idea of Koopman-Kalman Enhanced Variational Autoencoder (K^2VAE) is that a normal VAE struggles with long-horizon time-series because real-world series are non-linear and non-stationary, which makes the latent space hard to organize in a meaningful manner. This makes small prediction errors accumulate as the forecasting horizon grows. Therefore, the main idea of the architecture is to give the latent space the explicit structure of a linear dynamical system, so that the encoder no longer freely chooses μ and Σ, but instead tracks the state of something that is well understood [43].

The first part of this relies on Koopman theory, which states that non-linear systems can be lifted into a higher-dimensional space of measurement functions where its evolution becomes linear. This is governed by the Koopman operator K. The way K^2VAE implements this is through a KoopmanNet, where an MLP serves as the measurement function and K is approximated using the sum of a local part fitted with data and a learnable global part. This results in a linear latent space where transitions are simple matrix multiplications rather than tangled non-linear functions.

However, due to the imperfect linearization a KalmanNet is used to refine the result. Rather than having the encoder directly output the parameters of $Q(Z|X)$, K^2VAE runs a learnable Kalman filter over the linearized system, on which it alternates predict and update steps with learnable transition, control and observation matrices. This produces a refined state estimate and a covariance matrix at each step, which together become the mean and variance of the variational posterior $Q(Z|X)$. The latent uncertainty now has a concrete interpretation as the filter's belief about the system state, rather than being a free parameter the network has to learn from scratch.

Then the decoder acts as the inverse measurement function, mapping samples back to original space. This preserves the single-step generation of a standard VAE, avoiding the iterative sampling of diffusion- or flow-based forecasters, while the Koopman linearization handles the non-linearity and the Kalman filter mitigates error accumulation.

2.2 Hardware and Performance

There are many types of processing hardware and accelerators. In order to tell their efficiency and overall performance, there are many indicators.

2.2.1 FLOPS

Primarily used as a measure of computational performance, Floating-Point Operations Per Second (FLOPS) is a value that indicates how fast floating point operations

can be performed. It is primarily used to gauge peak hardware performance, but can also be used to infer how well an algorithm effectively utilizes the available hardware compute, though wall-clock timings are still preferred for direct algorithm comparisons. It has become an important metric for how fast a machine can perform ML training and inference [44]. It is worth noting that when it comes to hardware specifications, the number of FLOPS is usually the theoretical max for the given hardware, which is almost never obtained in real workloads.

When looking at FLOPS it is also worth noting what number of bits are actually used for the operations, as it has a large impact on performance. Going from FP64 (Floating Point with 64-bits of precision) to FP32 can more than double the number of floating point operations possible on the same hardware [45]. This trend carries over to other precisions, where the die-area required scales non-linearly with precision, decreasing the number of operations possible in a given timeframe at higher precisions. Common bit sizes for floating-point operations are 8, 16, 32, and 64 bits. In ML workloads BF16 (bfloat16) is also commonly used [46], trading mantissa bits for the same exponent range as FP32 to better preserve dynamic range during training.

Lastly, it is worth highlighting the difference between FLOPS and FLOPs. The first is the measure of how fast a given hardware can perform floating point operations, the second is the number of floating point operations performed. This is a crucial difference, as the first is a rate and the second conveys a total count, with the latter often being used to estimate the computational cost of a model independently of hardware. By the convention established in numerical linear algebra [47], a fused multiply-add (FMA) is counted as two FLOPs, one multiplication and one addition, even though it is executed as a single fused operation. This convention is also what vendors typically use when reporting peak FLOPS, and is worth keeping in mind when comparing reported figures across sources.

2.2.2 Memory Bandwidth

Memory bandwidth is the rate at which data can be transferred between the processor and memory, typically expressed in bytes per second (commonly GB/s or TB/s). It is one of the most important metrics for understanding real-world performance, since modern processors can perform arithmetic operations far faster than data can be supplied to them, leaving compute units idle whenever the memory subsystem cannot keep up [48], [49]. As with FLOPS, vendor-reported figures represent a theoretical peak that is rarely achieved in practice, and instead sustained bandwidth is measured empirically using benchmarks such as STREAM [50], which has become the norm for reporting achievable bandwidth.

A key strategy for mitigating this bottleneck on modern systems is the cache hierarchy, which also makes the bandwidth of a system difficult to characterize with a single number. The cache hierarchy hides memory access latency and increases effective throughput by introducing tiered memory with drastically different speeds and capacities [51]. This is effective because most programs repeatedly access the same data, so the effective bandwidth and latency for a given kernel tend to resem-

ble those of whichever cache level its working set fits in. It does, however, make performance highly sensitive to access patterns, where sequential and predictable accesses benefit from strategies like prefetching and thus see higher effective bandwidth, while irregular accesses do not [52]. As a rule of thumb, a repeatedly executed computation will be bound by the bandwidth of the smallest level of the hierarchy that can hold its working set [53], which is why empirical measurement is usually required to characterize real workloads.

2.2.3 Arithmetic Intensity

When the processor executes a program, there is some share of the executed operations that are memory operations. In order to compare how many data operations are done per one unit of data, the term Arithmetic Intensity, also called and interchangeable with the term OI, is used. It describes the ratio between the amount of arithmetic operations to how much data is read or written in memory, usually measured in FLOPs/byte [54], where the bytes typically refer to traffic to and from main memory, but can be measured against different cache levels.

Arithmetic intensity is largely a property of the algorithm and its implementation rather than the hardware it runs on, which makes it useful for reasoning about the cost of a workload independently of the machine that will execute it. As an example, dense matrix multiplication has a high arithmetic intensity, performing many arithmetic operations per byte of data loaded, which is part of why it maps so well to modern accelerators. In contrast, an operation like elementwise vector addition has very low arithmetic intensity, roughly one FLOP per several bytes loaded, and is therefore dominated by memory traffic rather than compute.

2.2.4 The Roofline Model

Building on the concepts of FLOPS, memory bandwidth and arithmetic intensity, Williams et al. [54] developed a visual model that can be used to easily determine whether a computation is memory- or compute-bound on a given piece of hardware. This is done by plotting attainable performance (FLOP/s) against arithmetic intensity (FLOP/byte), typically on log-log axes. The model exposes two ceilings, the peak arithmetic performance and the peak memory bandwidth. Together, these form a "roofline" shape, which can then be used to determine if a workload is compute-bound (hitting the FLOP/s ceiling) or memory-bound (lying on the bandwidth-limited slope). An example of a simple roofline can be seen in Figure 2.6.

Formally, the attainable performance of a kernel with arithmetic intensity I is given by $P(I) = \min(P_{\text{peak}},\ \text{BW}_{\text{peak}} \cdot I)$, where P_{peak} is the peak compute throughput and BW_{peak} is the peak memory bandwidth.

If a workload sits at the intersection of these two limits, a balance between bandwidth and compute is reached. This intersection is called the "ridge point", an example of which can be seen in Figure 2.7. The ridge point marks the minimum arithmetic

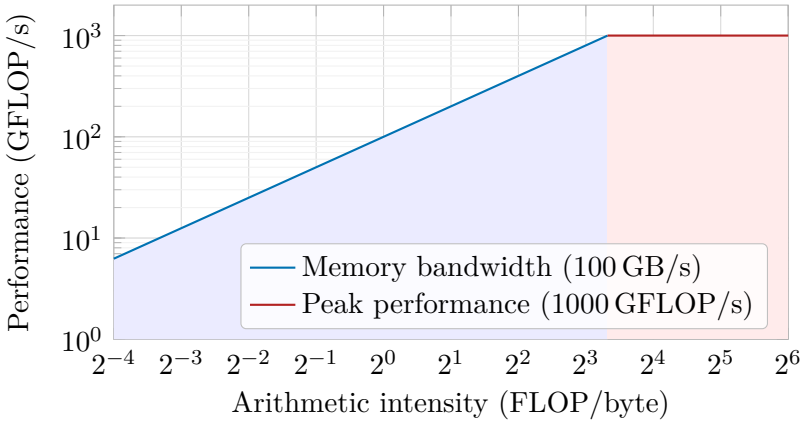


Figure 2.6: A simple roofline example, showing the compute against a single memory.

intensity at which peak compute performance becomes achievable: kernels to its left are memory-bound, while those to its right are compute-bound.

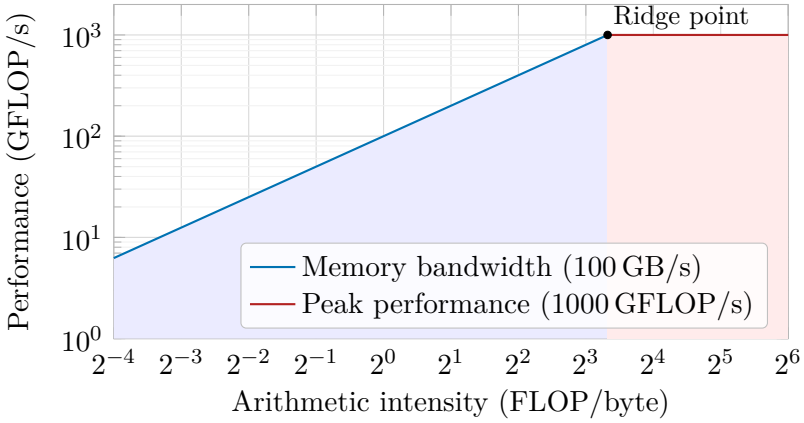


Figure 2.7: Simple roofline extended with a ridge point marker

However, on systems with a tiered memory hierarchy, the model is expanded with multiple rooflines. In this case, a slope is drawn for each tier of memory, creating multiple bandwidth ceilings. There are then multiple ridge points, depending on which level of the memory hierarchy dominates the kernel's data movement, as can be seen in Figure 2.8. These are helpful for understanding hardware utilization and how the application behaves in relation to the different memory bandwidths.

Beyond classification, the roofline model also informs optimization, memory-bound kernels benefit from techniques that raise arithmetic intensity (such as cache blocking or operator fusion), whereas compute-bound kernels are better served by improvements to instruction-level parallelism or vectorization.

2.2.5 Latency and Throughput

When doing a computation, especially when serving models, there are two general metrics that are of importance for speed. They are latency and throughput.

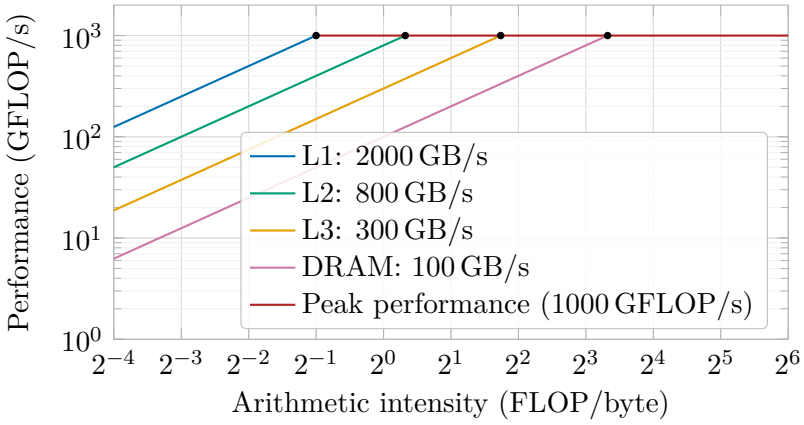


Figure 2.8: Roofline with multiple bandwidth slopes for different tiers of memory

The first of these, latency, is a measure of how long it takes between a request being submitted and the result coming back [55]. This determines how long a single prediction takes, and it is affected by factors such as model size, input complexity, and the underlying hardware. Because latencies are typically not uniformly distributed [56], it is common to report percentiles such as p50, p95, and p99 in addition to the mean, since the slowest requests often shape the overall behavior of a system more than the average does.

Throughput, on the other hand, is how much work the system can process during a certain period of time [55], typically measured in predictions per second. It is affected by model concurrency, batch size, and memory bandwidth, and it reflects the overall capacity of the system.

The two are typically in tension. Increasing batch size, for example, raises throughput by making better use of the hardware, but at the cost of higher per-request latency, since requests must wait to be grouped together [57]. The right balance depends on the workload, and in practice both metrics are reported because together they give a fuller picture of computational performance than either does alone.

2.2.6 Pareto Analysis

When optimizing for multiple objectives, there is often some amount of competition between them. Improving one objective may require sacrificing another, so rather than a single best solution, one ends up with a set of solutions that each represent a different trade-off. A solution is said to dominate another if it is at least as good on every objective and strictly better on at least one. The set of solutions that are not dominated by any other is called the Pareto-optimal set and the boundary they form in the space of objectives is called the Pareto frontier [58].

Dominated solutions are generally considered sub-optimal, since for each of them there exists an alternative that is better on at least one objective and no worse on any of the others. They can therefore usually be excluded when considering options. Only the solutions on the Pareto frontier are worth picking from, with the choice depending on which trade-off is most acceptable in the given context, for instance,

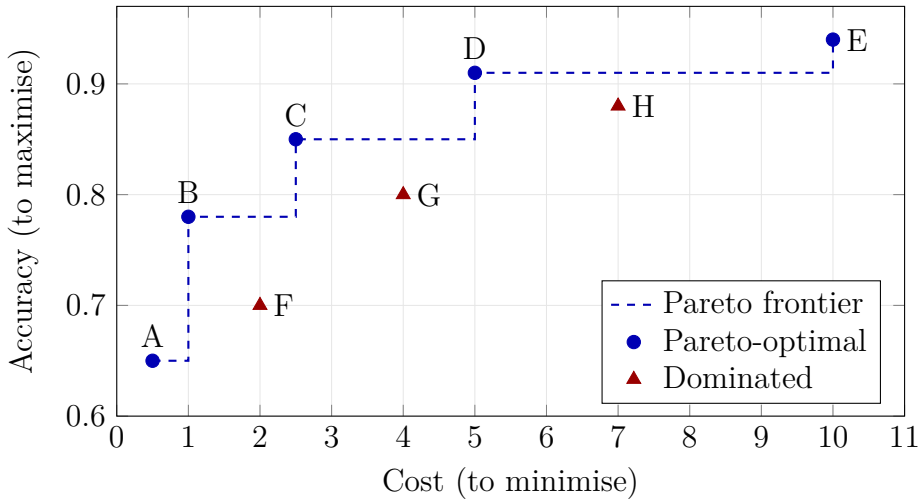


Figure 2.9: An illustrative Pareto frontier for a hypothetical problem with two competing objectives, cost and accuracy. Models A–E are Pareto-optimal and trace out the frontier, while models F, G, and H are dominated, at least one frontier model achieves both lower cost and higher accuracy.

how much accuracy one is willing to give up for lower cost or latency.

An illustrative example is shown in Figure 2.9, where eight candidate solutions are plotted in the space of two competing objectives. Five of them form the Pareto frontier, while the remaining three are dominated: for each, at least one frontier solution achieves both a lower cost and a higher accuracy.

2.3 Performance Profiling

To measure the performance characteristics of a program, metrics must be collected as it is running. This can be done through multiple methods, with various degrees of accuracy and complexity.

2.3.1 Hardware Profiling

One of the most direct methods of getting metrics for a program is to ask the hardware itself what it is doing. This can be done by measuring Performance Monitoring Counters (PMCs) [59], which track different events such as memory accesses, arithmetic instructions, cache misses, or instructions run. These values can then be used to calculate different statistics. In our use case, PMCs for determining arithmetic intensity is most interesting.

PMCs can be collected in two ways. Counting accumulates exact totals for a chosen set of events over a region of code, while sampling triggers an interrupt every N events and records the program state at that point, producing a statistical profile rather than exact totals. Sampling has lower overhead, but the recorded instruction pointer does not always match the instruction that caused the event, an issue known as skid[60].

There are multiple hardware profilers. For example, AMD μProf [61], Intel Advisor [62], and NVIDIA Nsight Compute [63] are vendor-specific profilers that are designed to measure their own hardware. All of these are able to measure roofline performance among several other metrics. Nsight Compute in particular is the standard tool for kernel-level GPU profiling and roofline analysis on CUDA kernels, while NVIDIA Nsight Systems [64] covers system-wide timelines such as kernel launches and host-device transfers. Third-party profilers are also available, such as LIKWID (Like I Knew What I'm Doing) [65], which can measure multiple architectures and compute types, including common CPU and GPU architectures.

The drawbacks of such profilers are that they require access to these hardware counters, which often both require specialized software and require elevated privileges to be able to access them. Another issue is that the number of physical counters is limited, typically only a handful per CPU core and even fewer concurrent metrics on GPUs. Measuring more events than fit in hardware requires multiplexing them in time and extrapolating [66], or in the GPU case, replaying the same kernel multiple times with different counter configurations [63]. Finally, doing proper profiling where you count every single event is often time-consuming and can massively increase the runtime duration, though with a high degree of accuracy [67]. Kernel replay on GPUs makes this overhead particularly pronounced.

2.3.2 Framework-Level Profiling

Some frameworks, such as PyTorch, provide their own profilers [68]. The core difference between framework-level and hardware-level profilers lies in what they instrument. Framework-level profilers target the framework itself and the workload running on it, while hardware profilers target the hardware. The framework does not need to ask anyone what happened during execution, because it knows it directly. Every operator dispatch and tensor allocation passes through code it controls, and can be traced directly. Framework-level profilers can additionally be enhanced with PMCs when lower-level data is desired, wrapping the hardware stack rather than replacing it. This is especially true on GPUs, where they often wrap various vendor Application Programming Interfaces (APIs) for improved metrics [69].

This difference also changes how FLOPs are reported. Since the framework knows each operator's type and the shapes of its input and output tensors, it can derive a FLOP count analytically from the operator definition alone. A matrix multiplication of shape $M \times K$ and $K \times N$, for example, is reported as $2MNK$ FLOPs [70]. Wall-clock time, tensor allocations, and host-device transfers are then measured separately and recorded alongside this estimate. The reported FLOPs therefore reflect what the operator is expected to compute rather than what the hardware actually executed [70], and the sophistication of the underlying derivation can vary greatly between profilers. Simple counters that handle only a few operator types sit at one end, while more sophisticated estimators can come close to the figures reported by hardware counters.

The drawbacks of framework-level profilers therefore differ from those of their hardware-level counterparts. They cannot explain why a given code path is slow, since events

such as cache misses are not visible at this layer [69], [71]. They also still incur a runtime cost, though lower than that of low-level profilers, and without requiring the elevated privileges or counter multiplexing discussed previously. Bookkeeping and serializing the collected data still take time, even if the mechanism differs from multiplexing or kernel replay.

2.3.3 Empirical Roofline Toolkit

Many types of hardware have known theoretical upper performance ceilings that can be obtained through manufacturer specifications, but they may differ from how it performs in reality. Sources of error may arise from several factors, such as runtime overhead or hardware differences from theoretical models. Instead of using theoretical performance ceilings when measuring workloads it is possible to instead use a tool that empirically finds performance ceilings.

The Empirical Roofline Toolkit (ERT) is a tool developed at the Lawrence Berkeley National Laboratory [72] that can run on x86 architectures as well as most GPUs. It runs custom micro-kernels with different configurations to empirically determine total max compute, and memory bandwidths at different tiers, such as cache layers and Dynamic Random-Access Memory (DRAM). Each kernel run has a given amount of floating-point operations per element that can be specified by the user.

This gives the benefit that ERT runs yield real-world performance limits for a machine, as workloads are executed much like how normal programs are run. When running other programs, their performance can be profiled and plotted against these limits, producing a comparison that is more “apples-to-apples”.

2.4 Slurm and Environment Modules

When running or executing tests, it can be a non-trivial task to allocate a machine, upload files if necessary, run code, and perform post-job tear-down. This is where the Slurm Workload Manager can be useful. It is built as a tool for management of clusters and to schedule jobs [73].

Slurm can be used to schedule and run workloads in a structured and well-defined manner. A job can have different resource allocations, such as CPU count or maximum DRAM allocation, but can also be run in exclusive mode to receive access to all the hardware that a machine can offer.

To ensure that an execution environment is as consistent as possible, one can use Environment Modules. This is a tool that allows for loading and unloading different types of “modulefiles” which contain operations that set up the shell for certain functionalities [74]. For example, one can set up a modulefile that initializes a complete NVIDIA CUDA environment, ready for immediate use without prior installation steps.

These two tools in combination simplify the process of environment setup, job scheduling, and final data collection.

3

Methods

To address the research questions and characterize the cost-latency trade-offs of deep learning forecasting architectures, the work was divided into three phases: literature synthesis and workload selection, profiling and benchmarking, and Pareto analysis.

3.1 Literature Review and Selection

This section describes the process by which candidate time series forecasting architectures were identified, evaluated, and selected for inclusion in the benchmark study. As the research questions require characterizing a Pareto frontier across fundamentally different architectural paradigms, ranging from recurrent networks to Transformer-based models, it was considered essential that the final model suite exhibit maximal variation in computational execution patterns rather than merely optimizing for reported forecasting accuracy. The selection process is therefore structured to ensure that each selected architecture exercises hardware resources in a qualitatively distinct manner, directly supporting the roofline characterization sought in RQ1 and the cross-architectural comparison required by RQ3. The subsequent section addresses the selection of hardware SKUs available within the Microsoft Azure cloud environment, against which the selected architectures are benchmarked.

3.1.1 Model Selection

The first phase of the study was a literature review done to find and identify reasonable current time series forecasting models. The purpose was to select candidates such that, for each computational class, the chosen model best represents the state of the art within that class. Since our research questions require comparing fundamentally different architectural paradigms against hardware limitations (RQ1), the selection was guided by maximizing variation in computational execution patterns rather than optimizing for forecasting accuracy alone.

To accomplish this, we first identified the most relevant computational classes, mapping our search to transformers, MLPs, RNNs, LSTMs, CNNs, MoEs, GNNs, and variational autoencoders. These classes were decided through a review of contemporary materials and articles, as well as the authors' own domain knowledge, and were chosen to represent as varied fundamental architectural types as possible. This ensures that each class exercises hardware resources in a qualitatively different way,

such as memory-bound sequential processing vs. compute-bound parallel attention. The list in its full is documented in appendix A.1.

Once the classes were established, the second part of the literature review took place. This phase focused on finding specific contemporary models for each category. We cast a wide net, creating a tracking document of all the relevant model architectures that we could find that were recent and/or mentioned frequently in the recent literature. These models were then categorized and had a brief description attached. This initial survey yielded 78 candidate architectures. Once this set had been compiled, we turned our attention to sifting through them.

The first criterion by which we filtered models was whether or not the source code was publicly available. This was chosen as an initial filter to keep the project within scope. We would not have the time to implement models from scratch, so restricting to open-source models reduced the time spent on implementation. We acknowledge that this may exclude recent proprietary or unpublished architectures. This also carries risks related to computability, and can affect the results since they may be implemented with differing knowledge of low-level performance characteristics.

The second criterion was whether or not the model natively supports multivariate data. This constraint was chosen based on our target data domain, namely ERP data, which contains many interdependent variables. Choosing multivariate-capable models also enables us to vary the number of input channels per architecture, directly supporting the investigation of hardware saturation behavior central to RQ2 and RQ3. Models that are fundamentally univariate were excluded, while models with straightforward multivariate extensions were evaluated case by case.

When these coarser filters were applied, we moved on to more fine-grained selection. To accomplish this, we created a 1–5 overall scoring system for evaluating the remaining candidates. These scores encompassed recency, citation frequency, and reported benchmark performance. It also took into account domain fit, assessing how well the architecture matches the characteristics of real-world ERP data. This resulted in a softer and less rigid approach than the hard filters, but was needed to sort through the remaining candidate architectures.

From this ranking, we selected only one or two model architectures from any given computational class (yielding a final selection of 8 models). The ranking ensured that only the models with the highest assessed potential and relevancy would be tested, while also providing a prioritized list of fallback candidates in case a selected model proved ill-suited during implementation. Where a higher-scoring candidate would have duplicated a class already covered by another selection, the next-best candidate from an uncovered class was preferred instead. This selection process ensures that the final model suite spans the computational diversity needed to characterize a meaningful Pareto frontier (RQ3), while remaining grounded in architectures that are considered current and competitive. The filtered-out models with their scoring can be found in appendix A.2.

3.1.2 Hardware Selection

Hardware was chosen based on Virtual Machine (VM) instances available in Azure CycleCloud. To get a representative selection, we chose two families: compute-optimized (F series) and GPU-accelerated (NC series). For each of these families, we chose two series to have a wider spread of hardware tested. In the compute-optimized family, we have one series of Intel CPUs (Fs_v2), and the other with AMD CPUs (Fas_v6), to increase spread across manufacturers. In the GPU-accelerated family, we have two series with different GPUs: NCas_T4_v3 and NCads_H100_v4 (including NCadis_H100_v4).

Within these families, we wanted to have a balanced selection that properly maps out memory-compute relationships. Therefore, we chose three VM sizes, where available, within each of the series. Generally, for each size, the amount of memory allocated to the machine is proportional to the amount of vCPUs given within that series. For example, series A with a VM size of 8 vCPUs has 16 GB memory, has in its next size 16 vCPUs and 32 GB memory.

There also exists limitations depending on VM family and series. This primarily includes hardware counters necessary to measure FLOPS for roofline analysis, limited to dedicated HPC-specific families, as the use of hardware counters in shared hardware can be used for side-chain attacks [75].

3.2 Dataset

This section covers the specific dataset used for the study. Though not the end goal of the study to maximize dataset accuracy, data shape/characteristics as well as pre-processing is still important for interpreting the final results. Therefore the dataset and relevant actions will be covered in detail in the upcoming sections.

3.2.1 Dataset Characteristics

Due to the restricted and sensitive specifics of the dataset, we avoid naming any exact numbers and instead describe the contents qualitatively. In this case, the specific ERP data revolves around individual agreements made with the company. Each monthly time step contains millions of rows, but the data only spans a bounded period of about 5–10 years. The dataset contains the length of the agreement, the key of the customer, and which specific product was purchased. Notably, it lacks any direct demographic variables about the customers, though it does contain hundreds of different product SKUs, making the product dimension extensive.

The dataset relates directly to individual subscription line-items at the customer-SKU level. These line-items include both single-activation agreements, such as a single purchase for a particular feature, and percentage-of-volume agreements, where some amount is paid for each item added by a customer. Because the raw data represents individual agreements, it tends to concentrate around yearly periods when most agreements are made and extended. Since the data lacks external demographic indicators to help predict behavior, capturing these historical patterns becomes even

more central. Renewals in particular result in seasonal, spiky data, since most renewals take place at the new year. The overall pattern can be visualized in Figure 3.1, though it is a fabricated graph showing the overall characteristics, not exact numbers.

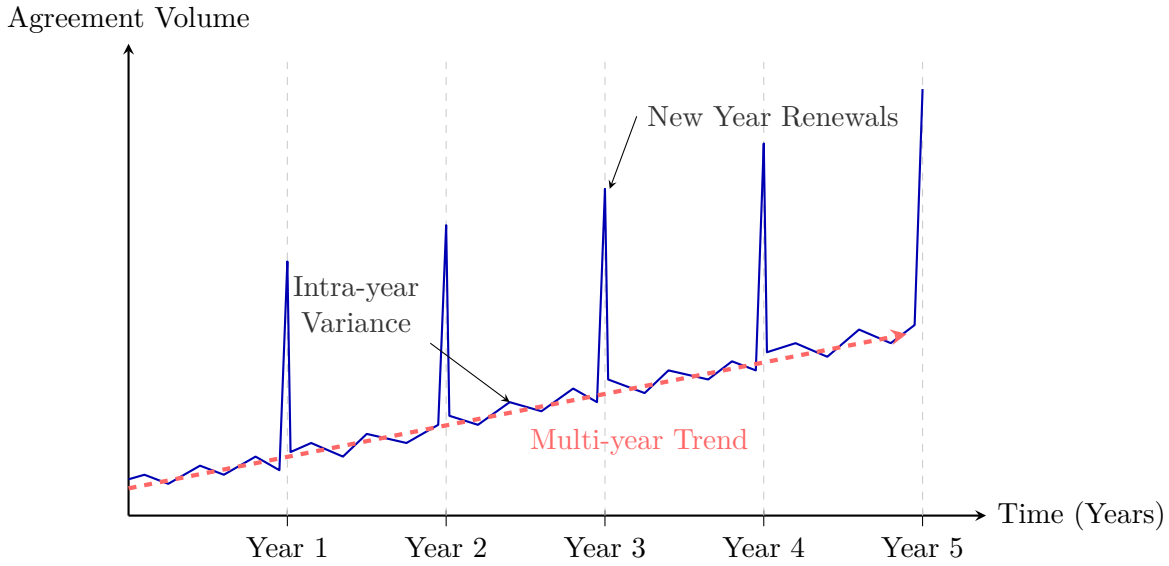


Figure 3.1: A qualitative representation of the underlying dataset characteristics. The diagram illustrates the long-running multi-year trend alongside substantial intra-year variance caused by continuous agreements. Most notably, it highlights the sharp, renewal-driven seasonality that concentrates at the start of each yearly period.

These aspects result in a dataset with both long-running trends, such as growth or loss over multiple years, and substantial inter-period variance, where individual licenses within a year can fluctuate considerably as a result of the percentage-of-volume and single-activation agreements described above. This increases the overall irregularity of the data and makes the underlying structure more complex to capture, motivating the exploration of deep-learning-based prediction tools.

These properties also make the dataset a useful benchmarking target. The combination of multi-year trend, sharp renewal-driven seasonality, high intra-year variance, absent demographic covariates, and a high-cardinality SKU dimension stresses the benchmarked models along several axes simultaneously, which is precisely the kind of pressure a meaningful comparison requires, and that would be hard-pressed to accomplish just using synthetic data.

3.2.2 Data Preprocessing

Due to some real-world issues with the data, such as duplicated rows and strange outliers, some data cleanup was done before proceeding with the normal pre-processing. Firstly, we deduplicated rows that referred to the same agreement, identified by agreement key and exact field-level matching. This removed clear spikes outside the expected renewal periods (e.g., the new-year peak), which could otherwise have

been mistaken for genuine seasonal signal. We also removed the most extreme outlier values that would otherwise disproportionately distort the overall curve.

Due to the nature of the data, containing a large volume of individual agreements for every time-step, an obvious pre-processing step was to aggregate individual time-step data. As a result, within each time-step, data points needed to first be grouped by their SKUs. These would then aggregate relevant data by summing things like total revenue. Agreements with a length that was more than one time-step were smoothed out such that each month in the total period had a revenue proportional to its share of that period. This linear smoothing is a deliberate convention applied uniformly across all benchmarked models, chosen for comparability rather than fidelity to any particular revenue-recognition rule. This gives a more realistic shape for the expected revenue of a single month than simply using the start of the agreement as an indicator. We also note that at this time, the models do not receive data such as new-year seasonality, fiscal markers or any such data, but expect to learn this from the sequence alone.

When training and tuning the models, we also used a train-validation split. Because the primary outcomes of this study are compute, cost and latency, we decided on an 80–20% split, where the start of the data period is used for training, and the end of the period is used for validation. The split is temporal only, SKUs are not partitioned between training and validation, so any SKU active across both periods appears in both, separated only by timestamp. This aligns with the forecasting task, which targets future revenue for SKUs already present in the catalogue rather than generalization to previously unseen SKUs. We did not use a train-validation-test split in this case because we wanted to use as much data length as possible for establishing variations in compute based on context window (see section 3.3 for the forecasting task definition). This was of great importance due to the data only reaching a few years back, and at a month-level accuracy any data we could use would be preferable. We acknowledge that this is a trade-off. This configuration cannot give a clean estimate of the out-of-sample accuracy on a truly new time-frame, but preserving data for the context-window sweep was prioritized.

The temporal-only split is also a natural fit for several of the models benchmarked, which were explicitly designed for panel forecasting across many related series with each series identity available as an input, a setting that a series-level holdout would undermine. With that in mind, we have tried to let each model accomplish pre-processing the way it originally did in the paper or source code, with all scaling statistics computed from the training split only to avoid leakage into validation. This means normalization and the encoding of the SKU dimension can differ across models, from learned embeddings to integer labels, implicit positional encoding via wide channel matrices, or no series identity signal at all. We saw this as a reasonable trade-off given that the study does not primarily focus on accuracy, but on evaluating each model in its "default" configuration for cost/latency trade-offs. We note that this is a confounding factor for accuracy comparisons, since models are not equivalently informed about series identity.

3.3 Forecasting Task

The models compared in this thesis are benchmarked on a single shared forecasting task. We define it once here so that the actual task done for benchmarking is clear.

The dataset prepared in subsection 3.2.2 is a panel of monthly revenues across N SKUs over T months. We denote it

$$X \in \mathbb{R}^{T \times N},$$

where $X_{t,n}$ is the aggregated revenue for SKU n in month t, computed according to the smoothing and aggregation rules described in subsection 3.2.2.

The forecasting task is to predict the next H months of revenue for every SKU given the most recent L months of history. At inference time, each model realizes a mapping

$$f_\theta : \mathbb{R}^{L \times N} \to \mathbb{R}^{H \times N},$$

where θ are the model parameters. Some models realize this mapping directly, while others are adapted to it through batched per-SKU invocations, as described in subsection 3.4.3.

Both L and H are treated as swept variables in the benchmark. The forecast horizon H ranges from 3 to 24 months, covering short-term operational forecasts as well as longer-range planning horizons relevant to enterprise financial planning. The context window L is swept from just a few months up to the full length of the training split T_{train}, allowing us to observe how each architecture scales as the input grows from a single seasonal cycle to the entire available history. The number of SKUs N is fixed across all runs and equals the full set of SKUs present in the prepared dataset.

Each model retains its native training loss, since rewriting the loss would alter the published architecture in ways that fall outside the scope of this work. To keep accuracy comparable across models, we evaluate every forecast with the mean absolute error,

$$\mathrm{MAE}(\hat{Y}, Y) = \frac{1}{HN} \sum_{h=1}^{H} \sum_{n=1}^{N} \left| \hat{Y}_{h,n} - Y_{h,n} \right|,$$

computed over the H-step forecast horizon and averaged across all N SKUs.

Because the primary outcomes of this thesis are computational rather than predictive, Mean Absolute Error (MAE) serves as a sanity check that the benchmarked models remain in a region where forecasts are meaningful, rather than as the optimisation target the work is structured around.

3.4 Benchmarking Pipeline

A custom benchmarking harness was developed in Python to automate the collection of performance metrics across the selected architecture and hardware pairs. Configuration for each workload was defined in a YAML file, which is read by the harness each time it is run. We used PyTorch as the primary way of running models.

To measure performance metrics, we use profiler tools to collect information during both model training and inference. In initial iterations, AMD μProf, Intel Advisor, Nvidia Nsight Compute, and LIKWID were used. However, it was observed during initial profiler testing that the majority of Azure VM sizes do not provide access to CPU hardware counters, and Nsight caused runtimes to increase to prohibitive lengths.

Seemingly, only Azure's HB family would allow access to necessary hardware counters. However, choosing this would restrict CPU profiling, eliminating the ability to select other CPU architectures. Due to these reasons, PyTorch's built-in profiler was chosen instead of these dedicated tools for both CPU and GPU workloads. While LIKWID could still have provided more accurate results for GPU runs, PyTorch was chosen for all workloads to ensure comparability regardless of hardware configuration used.

Using PyTorch's profiler, we collected data to estimate roofline metrics as specified in algorithm 1. This was developed with the profiler in mind, meaning that we relied on metrics from PyTorch as much as possible and minimized the amount of assumptions on top of this.

We measure the system by throughput, measured in inferences per second, which establishes the system's capacity. We also captured 50th, 95th, and 99th percentile latency measurements to analyze tail behavior critical for real-time operational constraints. Finally, we compare compute against DRAM throughput to identify the primary hardware bottleneck for each workload. To map the performance landscape, we swept over variables that can impact OI. These include batch size, input context window, and the required forecasting horizon. This approach ensures the benchmark captures how models behave under different data granularities.

To compare real-world data to synthetic data, we expanded the harness to generate and input a synthetic sine-wave series for models to execute on. This served as a very simple data shape that was not a constant value, or synthetic noise that could introduce sources of error when comparing hardware and models.

3.4.1 Benchmarking Environment

For the base environment, the entire harness runs on Python 3.12, using Lightning 2.6.1 and PyTorch 2.4.1. On GPU instances, CUDA 13.0 was used. PyTorch was limited to the chosen version due to uni2ts, a library used by Moirai-MoE, depending on PyTorch greater than or equal to 2.1, but less than 2.5.

Models were tested and scheduled as Slurm jobs under Alma Linux 9 running on Azure VMs through CycleCloud for Slurm. Upon scheduling new jobs, CycleCloud automatically allocated and then started resources necessary for running jobs. We created custom partition types and linked them to specific Azure SKUs which allowed us to efficiently run jobs on specific machines when needed. As allocated VMs are reserved to us, we ran jobs in exclusive mode to avoid under-usage of resources. Once all jobs for a specific SKU were finished, CycleCloud would after a timeout period de-allocate that VM and release its resources.

Algorithm 1: PyTorch profiler runtime metrics

Input: Function f to profile, Model M

Output: Total FLOPs (F_{total}), Total Bytes (B_{total}), Wall Time (T_{wall})

$T_{start} \leftarrow$ perf_counter();
Run $f()$ within `torch.profiler` tracking shapes, memory, and FLOPs;
$T_{wall} \leftarrow$ perf_counter() $- T_{start}$;
$S_{elem} \leftarrow$ default_element_size(M);
$F_{total} \leftarrow 0$;
$B_{total} \leftarrow 0$;
foreach *event e in profiler events* **do**
    $F_e \leftarrow e$.flops;
    $B_{in} \leftarrow \sum_{s \in e.\text{input\_shapes}}(\text{prod}(s) \times S_{elem})$;
    $B_{out} \leftarrow$ max(e.cpu_mem, e.device_mem, e.self_cpu_mem, e.self_device_mem);
    $B_e \leftarrow B_{in} + B_{out}$;
    if $F_e > 0$ **then**
        $F_{total} \leftarrow F_{total} + F_e$;
        $B_{total} \leftarrow B_{total} + B_e$;
    else if $e.name \in$ *MEMORY_ONLY_OPS* ***and*** $B_e > 0$ **then**
        $B_{total} \leftarrow B_{total} + B_e$;
return $F_{total}, B_{total}, T_{wall}$

Hardware rooflines were measured through ERT. This collected bandwidth for all available memory tiers, along with compute ceilings. Kernels were run at sizes 1, 4, and 64, with compute sizes 32, 256, and 4096 with float32 as the data type. The maximum DRAM working set memory was 1024MB (the size at which the working set would not fit in cache) using OMP_PLACES=cores and OMP_PROC_BIND=close. The working set growth was set at 1.02×.

Theoretical rooflines were derived for each memory tier and for peak compute. DRAM bandwidth was computed from vendor-published bus width, transfer rate, and channel count. L1 and L2 bandwidths were derived from per-cycle port widths and core counts, with microarchitectural parameters taken from vendor optimization manuals. For L3, a port-level upper bound was computed in the same manner, which does not capture interconnect contention or coherence traffic and therefore represents an optimistic ceiling rather than an achievable rate. Peak CPU compute was computed from core count, clock frequency and the FLOPs/cycle. For GPU nodes, peak compute and memory bandwidth were taken directly from vendor datasheets, which publish these values explicitly. These analytical values served both to validate the empirically measured ERT rooflines and to provide a comparison point against them.

If a model used too much memory, the operating system would sometimes kill it to free up overall system memory. To mitigate this, we added an option to run models as sub-processes. Should a model in this case use too much memory, and the system

kills it, the parent process would detect this and declare the model run as a failure. This was most useful when tuning models, as some hyperparameter combinations caused high amounts of memory use.

3.4.2 Model Harness and Porting

In order to run and interpret multiple models, we designed our custom harness to define a base interface for which each model would adapt to and run on. This would create a single layer for each of the models to work from, and also made it clear what functionality would be important for our models, by highlighting the most relevant methods from the start. Concretely, this base interface reduces to a `fit` and `predict` method, with model-specific parameters such as the input context length passed through a config dict at construction time rather than via dedicated setters. This let us flag early what models required deeper changes, and which ones only required surface level adaptations to fill out the relevant features.

Further, we needed to stabilize the actual data patterns of the models as well, to make sure that the output and input is consistent, thus further separating the harness from the models themselves. Inputs are wrapped in a `ForecastingData` object carrying the full series DataFrame, horizon, and validation window, and outputs are returned as a `ForecastingResult` containing the forecast DataFrame and a flag for confidence intervals. Batching is then left entirely to each model, `fit` and `predict` receive the entire dataset, and any DataLoader construction or per-series windowing happens internally. This made it possible to more easily adapt each model, as most specific required transformations could simply be done per-model as required. Furthermore, well structured outputs would make sure that we could compare the data more directly, as any model-specific changes could be handled and standardized.

The profiler attaches at this same interface boundary, wrapping the full `fit` and `predict` calls rather than instrumenting the model internals. This keeps the harness model-agnostic, but means the measured time includes data loading, Python dispatch, and result serialization alongside the model's arithmetic work, which is a caveat we return to when interpreting roofline results in section 5.2.

For the specific model implementation, we used the source code of the original paper or equivalent to a working iteration of the model. We started by making as few adaptations as possible to the original source code to make sure it could run our forecasting task. Once such a baseline was established, we started working on specific implementation that would be interface bound to our harness. By creating the initial baseline, we could verify if something in the adaptation influenced the results, and therefore check against regression on our ERP dataset. A risk of this approach is that engineering attention was not evenly distributed across models, those requiring deeper structural changes absorbed more implementation effort than those needing only surface-level adaptations, which may bias the comparison in their favor.

Worth noting is that we did not verify if the published versions of these models achieved the same scores on benchmarks as the ones shown in the associated paper, so any drift between the released code and the paper's reported numbers would

also affect our results. Additionally, this means we can only establish consistency between the minimally and maximally adapted model versions, and does not rule out issues introduced during the initial port. Any such issues would carry over into the end results, and a fuller validation against the original benchmarks is left to future work.

3.4.3 Multivariate Adaptation and Batching Strategy

The forecasting task is then fed into each of the models as to predict the SKU level revenue based on the historic data of all the SKUs. This creates a multi-forecast step, where a lot of different forecasts are made using its own and related data, which helps capable models learn cross-SKU patterns, at the cost of increased computational complexity. K^2VAE, Moirai-MoE, TimeFilter, TFT and TSMixer support this natively and process all SKUs jointly within a single forward pass. N-HiTS, TimesNet, and xLSTMTime do not, and were adapted to forecast each SKU individually, with the per-SKU invocations stacked along the batch dimension and dispatched as a single batched forward pass.

This adaptation was chosen to preserve each model's original architecture as published, limiting the changes to the data-loading layer rather than the model internals, and keeping the comparison anchored to each architecture's default configuration. The alternative, invoking the model sequentially once per SKU, would not only be unrealistic as a deployment scenario but would also conflate the architecture's intrinsic cost with Python dispatch overhead repeated N times. To verify this was the right trade-off, we ran early pilot measurements on a subset of the selected models comparing the sequential and batched variants, and observed roughly a 2–5$\times$ speedup from batching alone. We therefore adopted the batched per-SKU formulation as the default for all univariate-adapted models in the benchmark.

As illustrated in Figure 3.2, the batched per-SKU formulation does produce a computational profile that differs slightly from native multivariate forecasting, and this shapes how the roofline and Pareto results should be read. Joint forecasting shares activations and reuses weights against a wider effective tensor, raising OI, while the batched per-SKU variant replicates the model state across the batch and shifts the workload toward memory traffic.

For the multi-stream ERP setting that motivates this study, where forecasts are produced for many SKUs concurrently, the batched configuration is the more representative deployment scenario and is the basis on which we report results for N-HiTS, TimesNet, and xLSTMTime. For single-SKU or low-concurrency deployments, the same numbers are better read as an upper bound on achievable throughput, since the per-call latency without batching would be higher and the hardware less fully utilized.

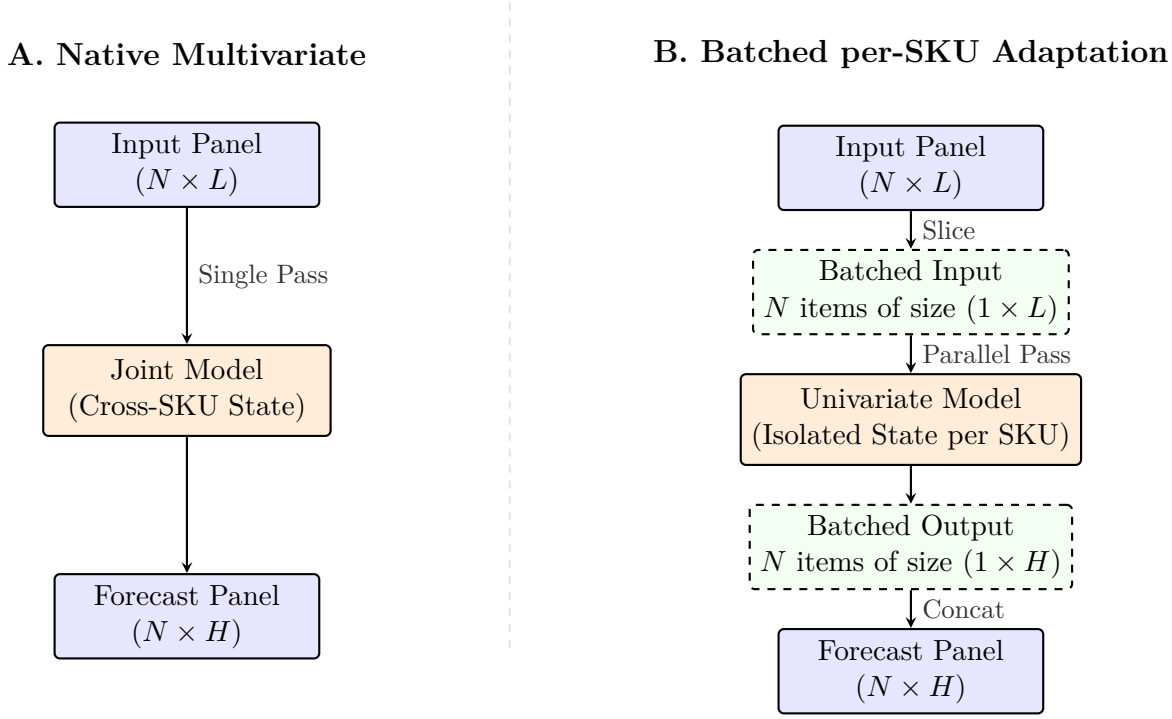


Figure 3.2: Comparison of execution patterns between natively multivariate architectures and adapted univariate models. Native models (A) process the full panel jointly, sharing state and raising arithmetic intensity. Adapted models (B) require slicing the input into independent sequences and relying on the batch dimension, which isolates the model state and shifts the hardware profile toward memory traffic.

3.5 Pareto and Roofline Analysis

In this section, we will cover anything on how the actual analysis was done. This covers cost calculations as well as Pareto dimensions, along with the interpretation and caveats for the roofline analysis.

3.5.1 Cost Calculations

We define the per-prediction inference cost as the product of the hourly on-demand price of the cloud instance and the average wall-clock latency of a single forward pass:

$$C(a, h) = p_h \cdot \frac{t_{\text{wall}}(a, h)}{3600} \tag{3.1}$$

where p_h is the hourly on-demand price in USD for hardware instance h (as listed in Table 4.2), and $t_{\text{wall}}(a, h)$ is the average wall-clock latency in seconds for architecture a running on h, taken over repeated measurement runs as described in subsection 3.4.2. The factor of 3600 converts the hourly rate to a per-second rate. A prediction here refers to one full forward pass of the model, producing forecasts for all N SKUs across the full H-month horizon simultaneously, as defined in section 3.3. The latency t_{wall} is measured at the same boundary as the profiler in subsection 3.4.2, wrapping the full predict call including any data processing, Python dispatch, and result serialization performed inside the model. This means the reported cost reflects the cost of serving the model as benchmarked, rather than the arithmetic work of the forward pass in isolation.

This formulation assumes steady-state serving and ignores training cost, which is treated as a one-time expense outside the scope of this work. It also attributes the full hourly price to the workload during the measured interval, which corresponds to a fully saturated instance running predictions back-to-back. Real deployments with bursty or scheduled forecasting workloads will see higher per-prediction costs in proportion to their idle time, so the values reported here should be read as a lower bound on serving cost rather than as an estimate of any specific deployment. Idle time and cold-start effects are excluded by construction, in line with the limitations stated in subsection 1.4.1. Storage and network egress costs are also excluded, they should be effectively the same, regardless of deployment.

We use on-demand pricing for the Azure region used by the partner company, as listed in Table 4.2 and retrieved on 2026-05-05. Pricing across Azure regions varies, and the absolute figures reported here are conditional on this snapshot. Spot and reserved pricing tiers are not analyzed separately, as the on-demand prices in Table 4.2 represent the upper bound on what a deployment would pay.

3.5.2 Pareto Dimensions

The Pareto analysis was conducted in two dimensions, cost per prediction against latency, with both axes derived from the same benchmark runs described in subsection 3.4.2. Cost per prediction was defined as in the previous subsection, and latency

was the average wall-clock time of a single forward pass over the measurement runs. Throughput was excluded as a frontier axis because, under the steady-state serving assumption already adopted for cost, it should reflect the latency for a fixed batch configuration and would add no information to the frontier.

The average latency was chosen because it characterized the steady-state behavior of the model under typical conditions, which aligned with the deployment scenario the cost calculation already assumed. The p99 latency was also computed for each configuration and considered separately, since tail behavior matters for latency-sensitive deployments but does not factor into the steady-state cost calculation. The p99 is not used to recompute cost, since cost is a function of how much hardware time is consumed in aggregate, not of the slowest individual request.

The various hyperparameters, such as context length, L, and horizon H were swept over separately. These create a separate parameter sweep graph, where each model is covered, and how the sweep itself affects the latency and cost.

Variance in the measured latencies was handled by use of repeated inference runs, where the average, and percentage latencies were recorded. The number of repeated measurement rounds for inference was 500, which was chosen due to the relative speed of inference, and wanting a large enough statistical figure to measure against. When deciding whether or not a point was dominated on the frontier, this variance is assumed to be handled by the averaging of the inference runs, and therefore any point that is within the frontier is considered to be dominated.

For the overall aggregation numbers, geometric mean was used. This as a way to lessen the impact of extreme outliers, and to make it more comparable across models.

3.5.3 Roofline

The roofline analysis interprets the raw profiler outputs from algorithm 1 as points on the empirical roofline established for each instance in Table 4.2. For a given run, OI was computed as

$$I(a, h, \phi) = \frac{F_{\text{total}}(a, h, \phi)}{B_{\text{total}}(a, h, \phi)} \tag{3.2}$$

and achieved performance as

$$P(a, h, \phi) = \frac{F_{\text{total}}(a, h, \phi)}{t_{\text{wall}}(a, h, \phi)} \tag{3.3}$$

where ϕ denotes a specific configuration of the swept hyperparameters (context length L and forecast horizon H), and F_{total}, B_{total}, and t_{wall} are the aggregated FLOPs, bytes, and wall-clock time recorded across all profiler events for that forward pass. An (I, P) pair is therefore produced for every (model, hardware, hyperparams) tuple in the benchmark.

To classify the given bound, we followed the rule that a workload is compute-bound when P approaches the peak compute ceiling, and memory-bound when it instead approaches the bandwidth slope at an intensity, I.

Each (I, P) pair was computed at the model level rather than the kernel-level, or per-layer level, to capture the model level as a single point. We summed F_{total} and B_{total} across all profiler events for the entire full forward pass. Since the forward pass of any given model can mix both high-intensity operations such as dense matrix multiplication with low-intensity operations, the result appear as a single average point whose position may not actually correspond to that of any individual kernel. We treat this as an inherent property of doing model-level roofline analysis rather than attempting per-operator decomposition, since the per-model point reflect what an operator sees when provisioning hardware for the full model, which aligns with the cost-latency framing of this work and with the architecture level granularity of RQ1.

Two interpretation caveats follow directly from how the underlying numbers were produced. First, the FLOPs counted by the PyTorch profiler are analytical estimates derived from operator shapes rather than hardware-measured counts, as discussed in subsection 2.3.2. The roofline therefore plots an analytical workload estimate against an empirical hardware ceiling, and a point close to the ceiling should be read as the model being close to that ceiling for the work the profiler attributed to it, not necessarily for the work the hardware actually performed. This was a required concession given in section 3.4.

The second caveat is that t_{wall} is measured at the same boundary as in the cost calculation, wrapping the full `predict` call rather than the model's arithmetic core alone. Time spent in non-core arithmetic is therefore included in P, which depresses achieved performance relative to a pure-arithmetic measurement. For the deployment-oriented framing of this thesis this is the correct boundary, but it does mean the roofline figures here are not directly comparable to per-kernel rooflines reported by tools like Nsight Compute.

In cases where the empirical roofline diverged significantly from the expected values, a theoretical roofline was constructed as a comparison point.

4

Results

4.1 Model Choice

The literature review yielded 78 initial candidate architectures. Applying the open-source availability filter reduced this to 59 candidates, and the multivariate-capability filter further narrowed the set to 28. From the remaining candidates, the scoring procedure described in subsection 3.1.1 was used to select one or two representatives per computational class, yielding a final set of 8 models. All first-choice selections were implemented without issue, and no fallback candidates were needed during implementation. Two of the models would later prove to be too slow to run across the full hardware set, as described in subsection 4.1.1.

The selected models and their corresponding computational classes are shown in Table 4.1. Most of these scored highly on domain relevance, as shown in appendix A.2. A few scored lower than alternative candidates, but were retained because they represented a class not otherwise covered, taking precedence over higher-scoring models that would have duplicated an existing class.

Table 4.1: Selected models and their corresponding classes.

Model	Computational/Model Class
N-HiTS	MLP
TSMixer	MLP
xLSTMTime	LSTM
TFT	Transformer
TimesNet	CNN
Moirai-MoE	MoE
TimeFilter	GNN
K2VAE	VAE

4.1.1 Models Excluded Due to Runtime

Two of the selected models, Moirai-MoE and TimesNet, ran too slowly to finish on every instance within the project timeframe. Early runs on part of the hardware completed without trouble, so the problem only became clear once the slower instances failed to finish in time. With some instances completing and others not,

there were too few shared configurations to compare these two models across the full hardware set, so they are excluded from the results. Because these were the only representatives of the MoE and CNN classes, the reported results cover five of the seven computational classes rather than all eight selected models. This narrows the cross-architectural breadth that the selection was designed to provide, and reduces the coverage available for the roofline comparison in RQ1 and the Pareto analysis in RQ3. All results and discussion should be read with this reduced coverage in mind.

4.2 Hardware Choice

The hardware selection yielded 11 Azure VM instances, organized into four groups based on the compute resources available. The first two groups consist of CPU-only instances spanning Intel Xeon and AMD EPYC processors at 8, 16, and 32 vCPU counts, covering a few-core to many-core range. The remaining two groups consist of GPU-accelerated instances, split between the low-end NVIDIA T4 and the high-end NVIDIA H100, with vCPU counts scaled to the number of attached GPUs. ARM-based instances and additional GPU generations were considered during selection but excluded from the final set, as noted in subsection 1.4.1. The selected instances and their hourly on-demand prices are shown in Table 4.2. The total cost spread across the selection ranges from $0.39/h for the smallest CPU instance to $18.15/h for the largest GPU instance, a range of roughly 47×.

Table 4.2: Chosen hardware instances. NVIDIA GPUs are assumed. Prices retrieved as of 2026-05-05.

Name	CPU	vCPUs	RAM **(GB)**	GPU	Cost/h **(USD)**
F8s_v2	Xeon Plat. 8370C	8	16	None	0.39
F16s_v2	Xeon Plat. 8370C	16	32	None	0.78
F32s_v2	Xeon Plat. 8370C	32	64	None	1.55
F8als_v6	EPYC 9004	8	16	None	0.52
F16als_v6	EPYC 9004	16	32	None	1.03
F32als_v6	EPYC 9004	32	64	None	2.07
NC8as_T4_v3	EPYC 7V12	8	56	1x T4	0.80
NC16as_T4_v3	EPYC 7V12	16	110	1x T4	1.28
NC64as_T4_v3	EPYC 7V12	64	440	4x T4	4.60
NC40ads_H100_v4	EPYC 9V84	40	320	1x H100	9.07
NC80adis_H100_v4	EPYC 9V84	80	640	2x H100	18.15

4.3 Baseline Performance

As seen in Table 4.3, we found that in CPU performance, core count had a linear relationship with compute performance and L1 bandwidth inside each series. However, comparing GPU nodes to CPU nodes, we observed a downtick in compute

and a large decrease in L1 bandwidth. Comparing GPU nodes to each other, both compute and bandwidth scales more in line with hardware tier.

Table 4.3: Achieved empirical hardware roofline performance

SKU	FP32 Compute (GFLOP/s)	L1 Bandwidth (GB/s)	LLC Bandwidth (GB/s)
F8s_v2	368.50	726.17	144.19
F16s_v2	737.84	1451.94	167.27
F32s_v2	1459.65	2910.31	476.74
F8als_v6	933.53	2560.76	142.53
F16als_v6	1851.35	4371.05	290.67
F32als_v6	3634.15	9531.10	522.5
NC8as_T4_v3	694.06	321.66	177.96
NC16as_T4_v3	692.04	320.92	178.08
NC64as_T4_v3 [1]	692.11	321.38	177.29
NC40ads_H100_v4	1250.23	446.84	178.03
NC80adis_H100_v4 [1]	1248.88	447.95	177.9

When comparing empirical numbers with theoretical ones, as seen in Table 4.4, we observed a large discrepancy between theoretical and empirical numbers. Empirical numbers were drastically lower than theoretical on GPU nodes, but close to theoretical values in the Fals series and Fs series, at least in relation to compute.

Table 4.4: Theoretical hardware roofline performance

SKU	FP32 Compute (GFLOP/s)	L1 Bandwidth (GB/s)	LLC Bandwidth (GB/S)
F8s_v2	435	870	440
F16s_v2	870	1740	870
F32s_v2	1740	3480	1740
F8als_v6	950	2800	950
F16als_v6	1900	5700	1900
F32als_v6	3800	11300	3800
NC8as_T4_v3	8100	4070	815
NC16as_T4_v3	8100	4070	815
NC64as_T4_v3	16200	8140	1630
NC40ads_H100_v4	66900	33000	12000
NC80adis_H100_v4	133800	66000	24000

[1] Multi-GPU instance. ERT does not run on multi-GPU environments, so single-GPU was benchmarked

4.4 Hardware Resource Utilization

TFT had a considerably higher achieved FLOPS than the other models with N-HiTS achieving the highest OI. Additionally, NC64as_T4_v3 showed very low compute compared to other SKUs, and NC80adis_H100_v4 would occasionally show this same behavior. Apart from the other models, TFT also had a wide discrepancy in OI between CPU and GPU models. Note that due to the aggregation of multiple models and systems in the same chart, no roofline is shown since it would be hardware bound. Instead treat this graph as a way to picture the relative computational behavior across multiple hardware SKUs.

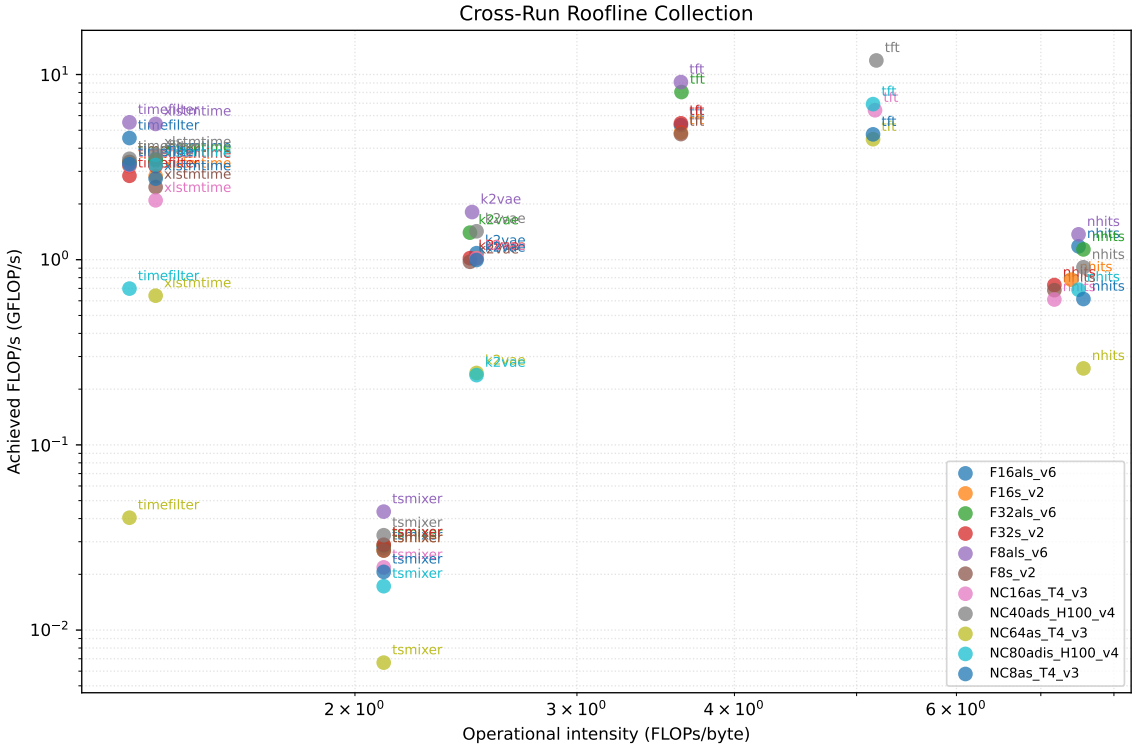


Figure 4.1: Comparison between models and SKUs for roofline

Per-hardware rooflines give a clearer picture of where each model sits against the compute and bandwidth ceilings. The models occupy a similar range of operational intensity on every machine, while their position relative to the ridge point varies with the hardware. Figure 4.2 illustrates this for the F8s_v2 and Figure 4.3 for the NC40ads_H100_v4.

The operational intensities of the models stay in a similar range across machines, but the ridge point itself shifts considerably between hardware tiers. In Figure 4.2, the ridge point sits at roughly 4 FLOP/byte, while in Figure 4.3 it shifts to roughly 15 FLOP/byte. This follows from the H100 pairing far more compute with comparatively less DRAM bandwidth than the CPU, as also reflected in the empirical values in Table 4.3.

The shift means that the same workload can fall on opposite sides of the ridge depending on the machine. N-HiTS, for example, sits at an operational intensity of

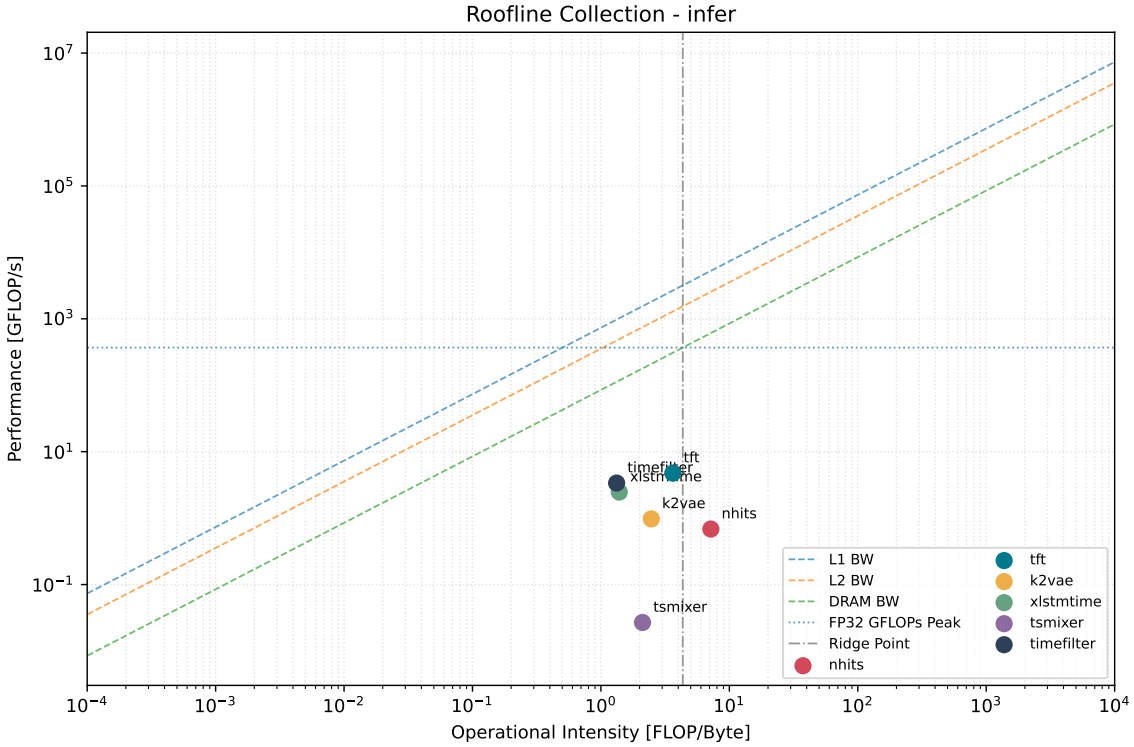


Figure 4.2: Model collection roofline for the F8s_v2 node.

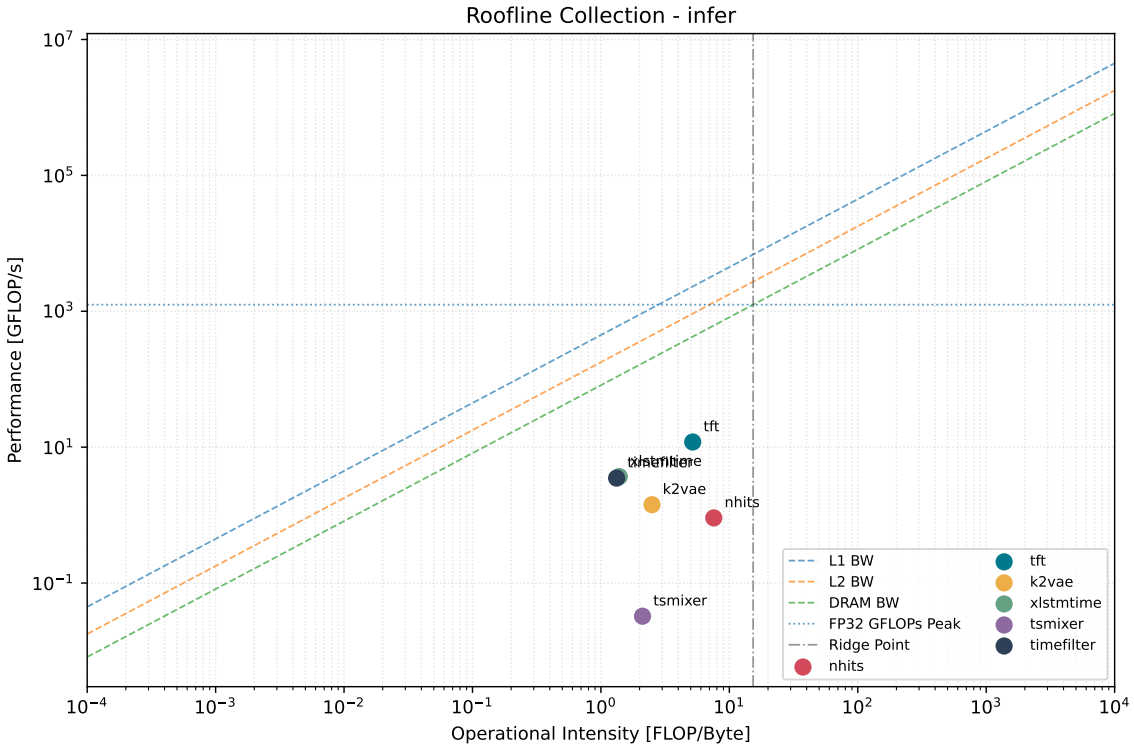


Figure 4.3: Model collection roofline for the NC40ads_H100_v4 node.

around 7 FLOP/byte in both Figure 4.2 and Figure 4.3. On the F8s_v2 this places it past the ridge point, in the compute-bound region. On the NC40ads_H100_v4

the same point falls well to the left of the ridge, putting the workload in the memory-bandwidth-bound region. The other models follow a similar pattern, all of them are bandwidth-bound on the H100, while several of them sit closer to the compute-bound side on the CPU.

Being on the compute-bound side of the ridge does not mean a model approaches peak compute. Across both Figure 4.2 and Figure 4.3, every model lands several orders of magnitude below the relevant ceiling. On the F8s_v2 the fastest models reach a few GFLOP/s against an empirical peak of roughly 370 GFLOP/s. On the NC40ads_H100_v4 the gap widens further, with the same models reaching around 10 GFLOP/s against a peak of roughly 1250 GFLOP/s.

Because the empirical ERT measurements diverged substantially from theoretical values on GPU nodes, we also constructed a theoretical roofline for the NC40ads_H100_v4, shown in Figure 4.4. As listed in Table 4.4, the theoretical peak compute of around 66900 GFLOP/s sits roughly 50 times above the empirical ceiling of 1250 GFLOP/s. This widens the gap between the models and the relevant ceiling, and shifts every model further left relative to the ridge point. The implications of this divergence are discussed in subsection 5.2.1.

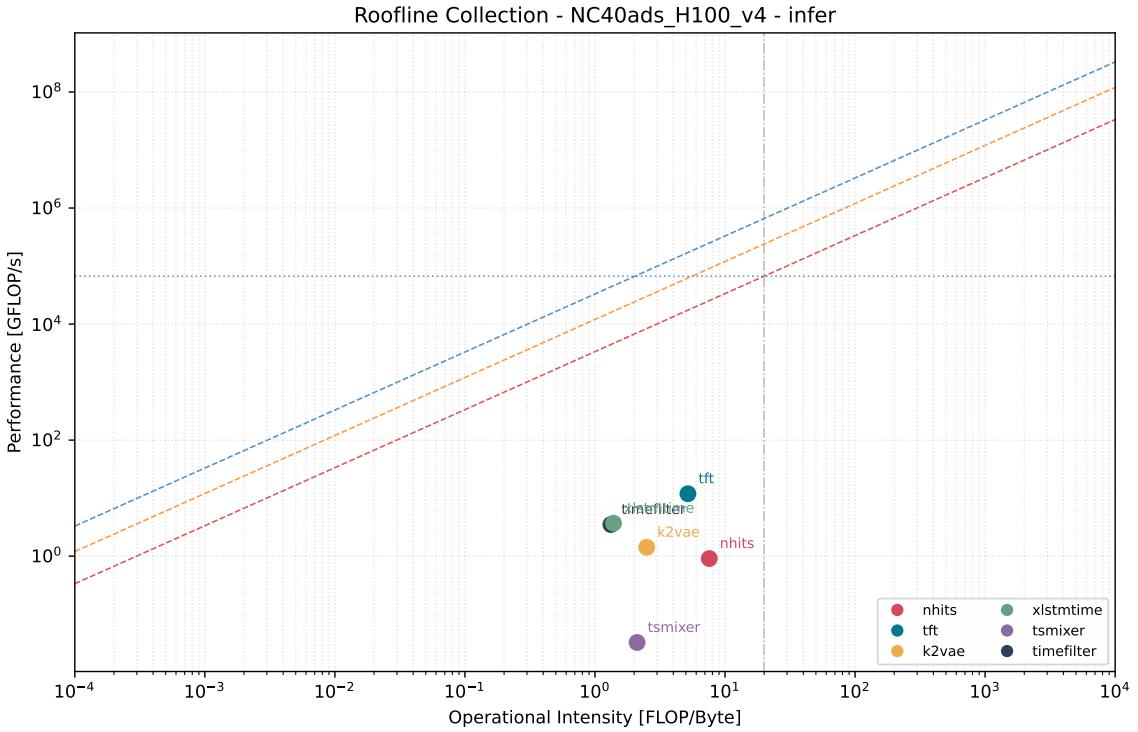


Figure 4.4: Model collection roofline for the theoretical roofline of the NC40ads_H100_v4 node.

These per-hardware rooflines clarify which ceiling each model is bound by, but the achieved performance also reflects factors outside the roofline itself, including the profiler boundary discussed in subsection 3.5.3 and the analytical nature of the profiler FLOP counts.

The clustering of models in operational intensity also obscures a wide spread in

achieved performance. TSMixer in particular sits roughly two orders of magnitude below the other models in both Figure 4.2 and Figure 4.3, at around 0.05 GFLOP/s. The remaining models cluster between roughly 1 and 10 GFLOP/s. This spread is consistent across machines and is not driven by any single hardware tier.

A full list of rooflines can be found in Appendix D.

4.5 Real-World Divergence

In general, as seen in Figure 4.5, latency and FLOP/s ratios remained somewhat consistent between SKUs, with latency being higher in real-world than synthetic runs, and FLOP/s being lower in the real-world than synthetic. However, NC64as_T4_v3 showed a large discrepancy to this trend, with considerably higher FLOP/s in real-world and lower latency compared to synthetic runs.

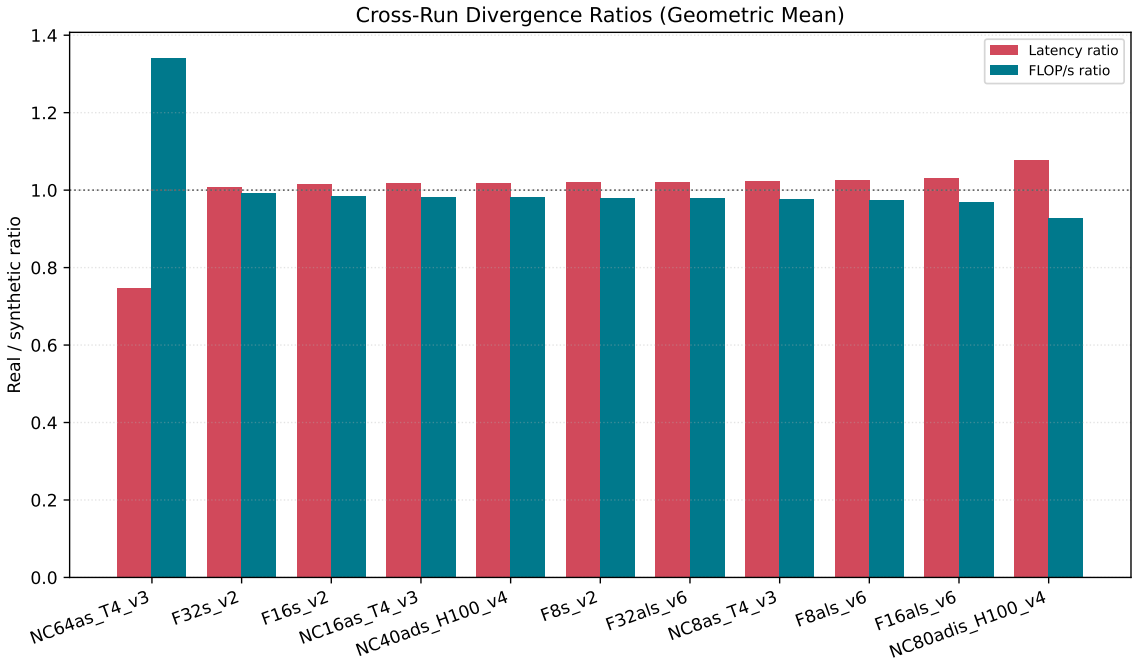


Figure 4.5: Real data vs. synthetic data divergence between SKUs

When comparing models in Figure 4.6, we saw much of the same trend as in Figure 4.5. Breaking this trend is TFT, which had a ratio near 1.0 for both metrics. There was also tsmixer and xlstmtime, both of which presented similar results to NC64as_T4_v3, albeit with smaller extremes.

4.6 Scaling and Compute

To see how runtime behavior scales with configuration, we swept three model parameters. These were batch, context (the number of previous data points) and horizon (the number of new points to predict). As in previous sections, this is an overview

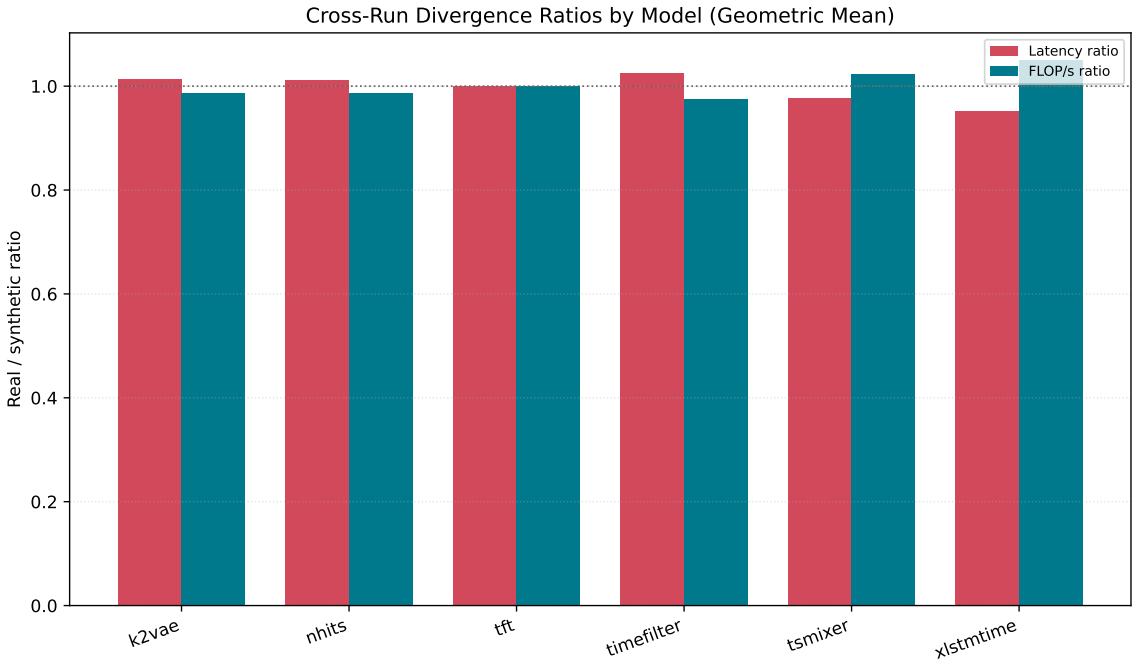


Figure 4.6: Real data vs. synthetic data divergence between models

of the data with representative examples and outliers. The remaining tables can be found in Appendix B.

The first parameter, batch size, had essentially no effect on inference OI when context and horizon were fixed. For example, in TSMixer (Table 4.5), batch values from 16 to 512 at fixed context and horizon produce the same inference OI. However, FLOP/s does respond to batch size, generally rising as batch grows and reflecting better hardware utilization.

For context, the typical pattern is visible in TSMixer (Table 4.5). The general trend seems to be that as the context grows, so does the inference OI. As context grew from 12 to 48, the inference OI increased from 2.10 to 3.96. The same direction held for K^2VAE, N-HiTS and xLSTMTime, though the magnitudes could differ. For example, xLSTMTime stayed in a narrow band from 1.39 to 1.50. All of these also show a corresponding increase in achieved FLOP/s as OI rises.

There was one exception to this trend, namely TimeFilter (Table 4.6). For this model, the trend reversed, with inference OI decreasing as context grew. As context grew from 4 to 48, OI fell from 1.32 to 1.02. Interestingly, the training OI rose sharply over the same sweep, from 35 to over 200, behaving like the inference OI of the other models.

Lastly, horizon had a relatively small effect on inference OI for most models. In the TSMixer table, OI varies only between 3.04 and 3.08 across horizons from 6 to 24 at a fixed context of 24. The only exception is K^2VAE, where horizon meaningfully shifts inference OI, from 1.71 at horizon 3 up to 2.92 at horizon 24 for similar context.

TSMixer's inference FLOP/s also remained orders of magnitude lower than the other

Table 4.5: Characteristics across trials for TSMixer

Batch	Ctx.	Hor.	Train OI	Train GF/s	Infer OI	Infer GF/s
32	12	6	3.75	15.20	2.10	0.03
256	18	6	9.69	31.55	2.65	0.04
128	18	6	9.69	32.79	2.65	0.04
32	18	6	5.17	20.50	2.65	0.05
32	12	3	3.56	14.92	2.10	0.03
32	24	6	6.37	24.24	3.04	0.06
512	30	3	13.86	57.91	3.33	0.07
32	30	6	7.41	32.47	3.34	0.06
16	24	18	4.02	16.40	3.07	0.05
64	42	12	17.10	45.97	3.78	0.10
64	36	12	15.63	53.70	3.59	0.09
256	24	18	12.07	36.27	3.07	0.06
16	30	18	4.67	17.54	3.37	0.06
128	42	18	17.13	59.05	3.80	0.09
512	12	24	7.21	13.86	2.12	0.03
256	48	18	18.42	58.55	3.96	0.10
256	24	24	12.13	29.90	3.08	0.06

Table 4.6: Characteristics across trials for TimeFilter

Batch	Ctx.	Hor.	Train OI	Train GF/s	Infer OI	Infer GF/s
32	10	3	35.00	29.80	1.32	2.94
32	8	3	35.00	30.35	1.32	2.99
32	4	3	35.00	30.14	1.32	2.75
32	12	3	35.00	30.14	1.32	3.12
32	16	3	54.21	33.87	1.23	4.39
512	10	3	55.78	35.84	1.32	3.07
32	4	12	42.23	30.56	1.32	3.38
32	10	12	42.23	29.85	1.32	3.13
32	8	12	42.23	30.48	1.32	3.39
16	14	18	24.53	23.78	1.33	2.71
32	14	18	49.07	29.24	1.33	2.69
128	28	3	142.07	30.59	1.11	8.58
512	36	6	170.77	30.36	1.06	7.96
512	38	6	170.77	30.24	1.06	9.19
256	38	6	170.77	30.90	1.06	8.68
64	24	24	120.19	32.64	1.14	7.45
32	48	3	200.48	30.38	1.02	9.72
256	28	24	137.96	27.07	1.11	6.49
128	36	24	166.43	28.47	1.07	9.05

models across all tested configurations, indicating that this is a consistent property of the model rather than something tied to a specific operating point. For TFT, all sweep changes had relatively small effects, producing a narrow inference OI band from 3.56 to 3.67, a total range of 0.11.

4.7 Cost-Efficiency

This section primarily covers the geomean results across all tested model configurations per machine. Individual model results are covered where interesting, with the full per-model breakdown available in Appendix C.

The median case for each machine is shown in Figure 4.7. For this collection of models running on this particular workload, the figure indicates a clear latency advantage for GPU-equipped machines over CPU-only instances. The Pareto frontier at the p50 level consists of NC40ads_H100_v4, NC8as_T4_v3 and F8s_v2, while F16s_v2, F32s_v2, F8als_v6, F16als_v6, F32als_v6 are dominated by at least one frontier point on both axes.

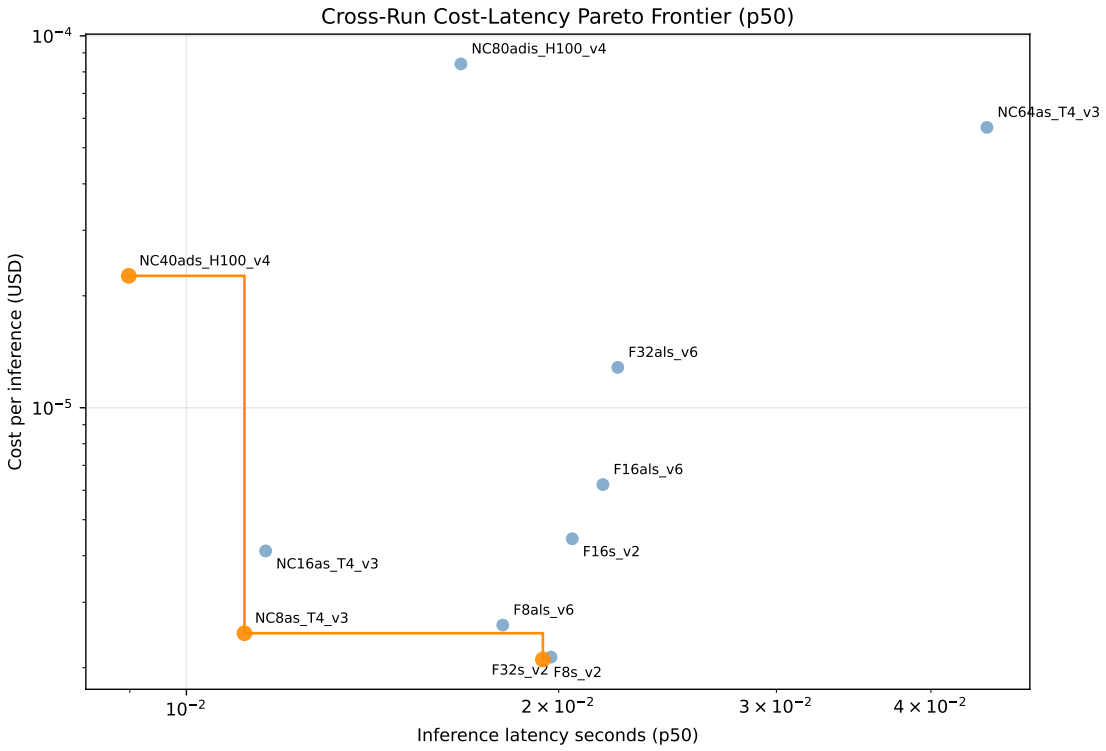


Figure 4.7: Cost-latency across models, p50

The lowest p50 latency in the geomean was achieved by NC40ads_H100_v4, which as seen in Table 4.2 pairs an H100 with an EPYC 9V84 across 40 vCPUs. This is the newest hardware SKU tested and the most high-end. This lead did not hold at the tail, however. In Figure 4.8, NC8as_T4_v3 took the latency lead despite using a cheaper T4 and only 8 vCPUs. One possible reading is that the smaller instance exhibits tighter runtime consistency for this workload, though the underlying

cause is left for the discussion. The frontier membership at p99 shifts slightly, with NC40ads_H100_v4 leaving the lowest latency spot in favour of NC8as_T4_v3.

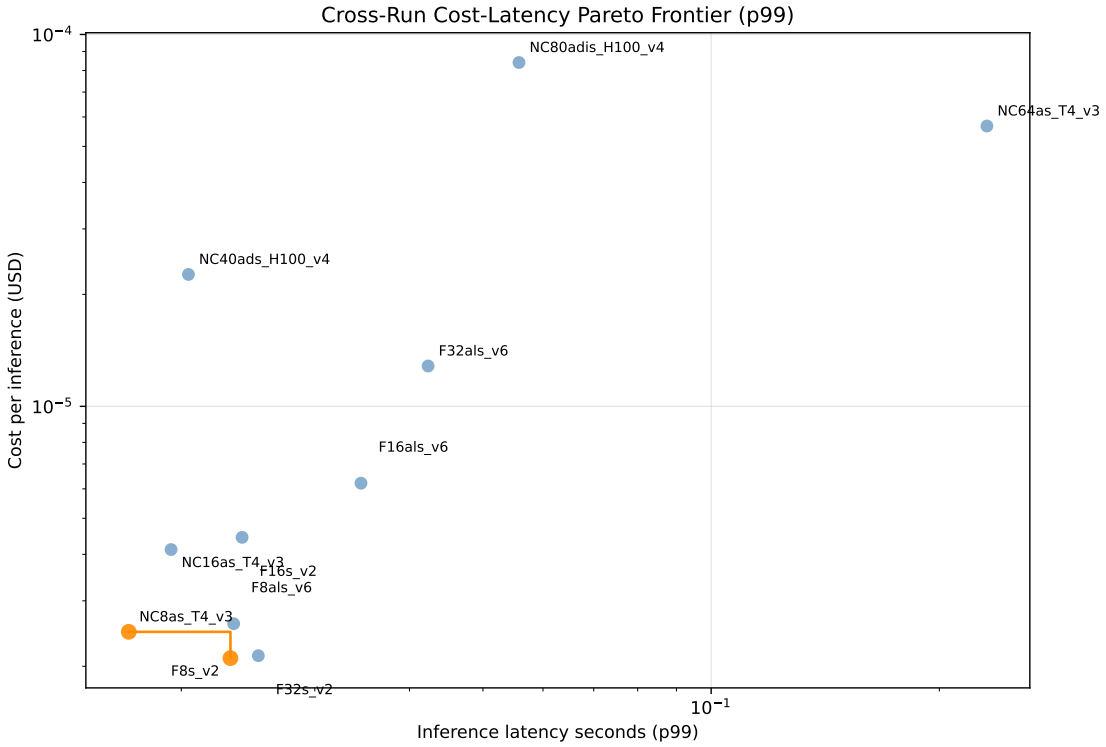


Figure 4.8: Cost-latency across models, p99

From a cost perspective, F8s_v2 occupies the low-cost endpoint of the frontier in both the p50 and p99 cases, achieving a lower cost per inference than the larger F-series CPU instances while remaining competitive on latency and tail behavior in the geomean, some reasons for which will be discussed later.

When looking at the models individually, they mostly follow the same pattern as the geometrical mean shown in Figure 4.7, but larger emphasis on GPU-based systems on the frontier. Some models, however, had significantly lower latencies on CPUs as compared to GPUs, in particular TSMixer has all GPU-based nodes dominated by the F8s_v2 node. This applies in both the p50 and p99 cases, shown in Figure 4.9 and Figure 4.10 respectively. Similar findings were also found for TimeFilter as seen in Appendix C.4.

Contrarily, the opposite relationship between GPU and CPU performance was also found for some models, particularly for TFT and N-HiTS. Where GPU nodes both outperform and dominate the CPU nodes, both from a cost and latency perspective. For N-HiTS these results can be visualized in Figure 4.11 and Figure 4.12.

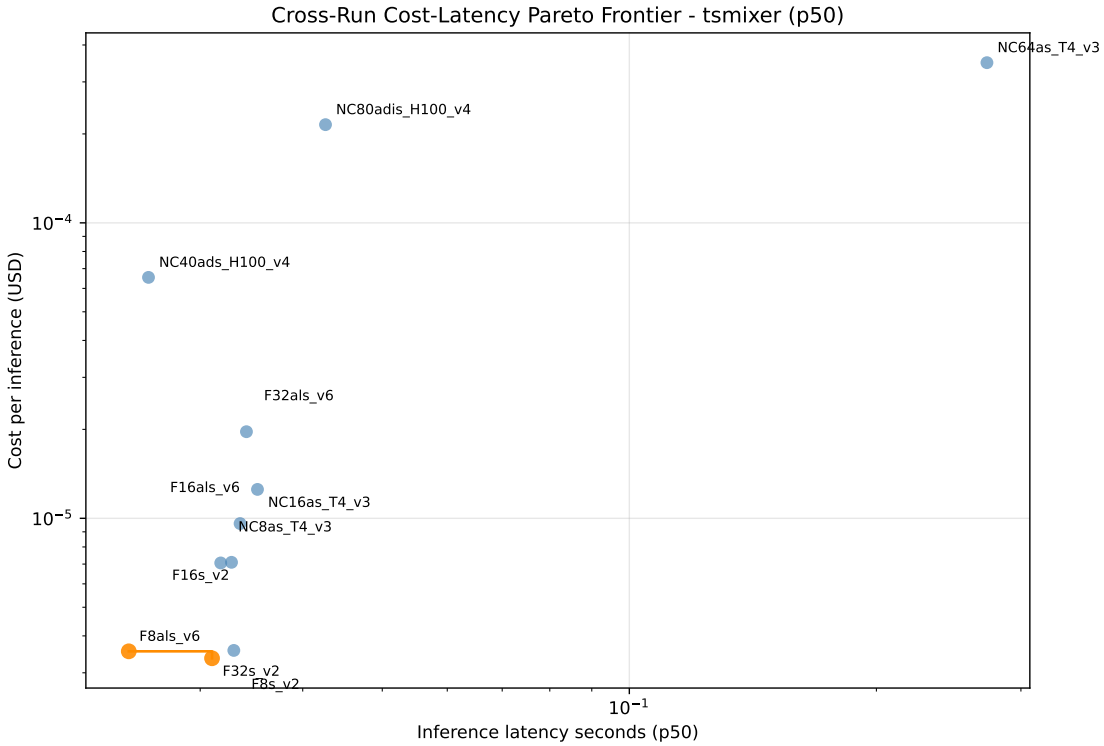


Figure 4.9: Cost-Latency for TSMixer, p50

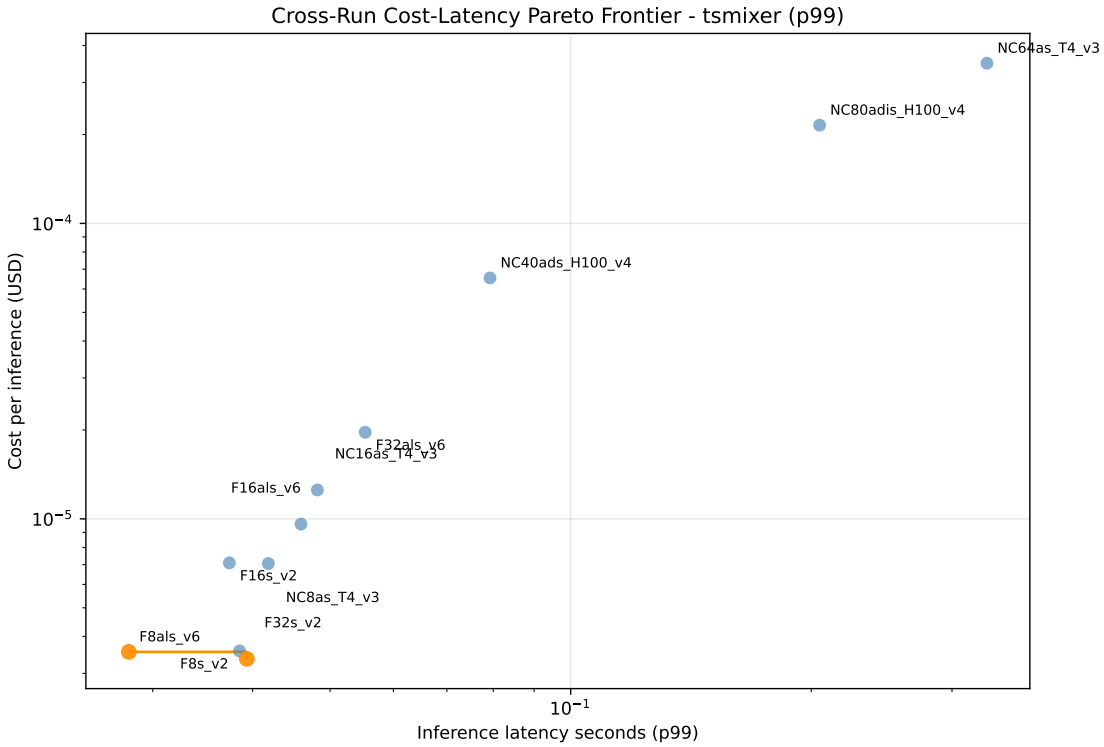


Figure 4.10: Cost-Latency for TSMixer, p99

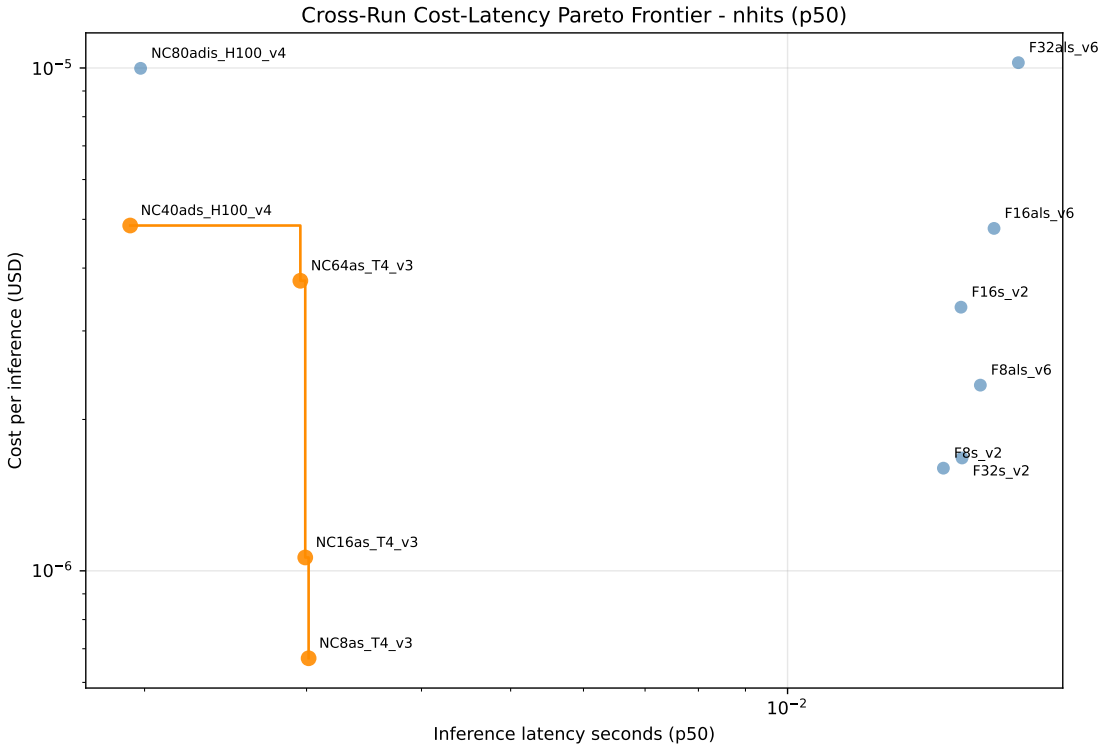


Figure 4.11: Cost-Latency for N-HiTS, p50

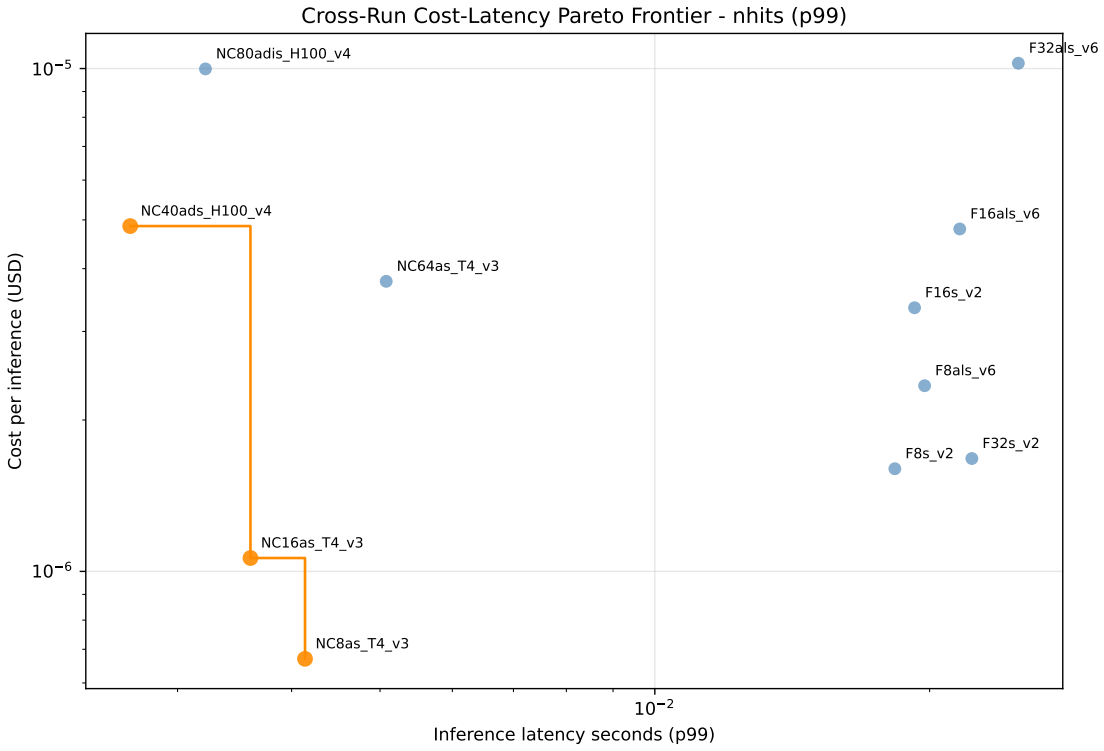


Figure 4.12: Cost-Latency for N-HiTS, p99

5

Discussion

This chapter presents the analysis for the report. The text is divided into four sections. Section 5.1 discusses the rooflines and resource utilization anomalies and tries to explain some of the results from the perspective of potential sources of errors. Section 5.2 covers scaling behavior and some outliers. Section 5.3 focuses on the cost-latency trade-offs and efficiencies based on our results. Lastly, 5.4 will discuss the impact of synthetic versus real-world data for our use case.

5.1 Exclusion of Models

Due to the previously mentioned exclusion of Moirai-MoE and TimesNet, we expect that our results look different from what they would have with those included. Their long execution times could likely make them differ in computational characteristics from other models that ran in more comparable times. The way the characteristics differ could depend much on what made these models run slower than the others, which directly impact FLOPS and OI. For example, they may severely under-utilize hardware and simply appear far downwards in the roofline model, or they may also have data or compute patterns that much saturate memory or compute, but leads to an overall slowdown.

5.2 Rooflines and Resource Utilization

This section interprets the roofline and resource utilization results from section 4.4. We first address a methodological concern about the empirical GPU ceilings, then discuss what the collective roofline picture reveals about hardware and model utilization, then lastly follow it up with some discussion on specific models.

5.2.1 ERT and GPUs

As we start looking at the rooflines and the utilization of the target nodes, there are various interesting trends. For example, one of the most noticeable things is that while the ERT values for the CPUs are rather close to the theoretical numbers for the target hardware, the same cannot be said for GPUs, which are drastically lower. While the empirical results for the given hardware are expected to be lower than the theoretical bounds, it is not expected for them to be an order of magnitude

lower, such as for the H100, which is about 50× slower in compute for the empirical measure (Table 4.3) compared to the theoretical (Table 4.4).

There are a couple of possible reasons for these results. The first is that the ERT does not properly exercise the GPU, even though the workload does run on it. We confirmed via `nvidia-smi` that the GPU was active during the measurements, so the issue is not that the workload silently fell back to the CPU. Instead, the ERT could be running a workload on the GPU that is not properly tuned for it, for example by relying on targeting stronger cores rather than thousands of small ones, or by not scaling the compute requirements in the same way that it does for the CPU measurements. This would heavily skew the results by creating a parallelization mismatch, and it would also affect the memory usage, where CPUs are expected to have larger local caches than GPUs, potentially memory-starving the whole measurement.

The second possible reason is that the nodes themselves were wrongly configured, and therefore were unable to fully utilize the provided GPUs. If that were the case, that would affect the rest of the results for latency on the Pareto frontier. By underutilizing the GPU-based nodes, it would skew the results in a way that drastically increases the final latency, thus making them become dominated by other nodes that would otherwise be drastically slower. This is in addition to making the gap potentially seem smaller than it could be otherwise, making lower-cost CPU nodes seem more latency-competitive than they really are.

Of these two, we find the first more likely. The strongest piece of evidence is that the same GPU produced essentially the same ERT result on nodes with very different CPU counts, as can be seen by comparing the T4 results on NC8as_T4_v3 and NC16as_T4_v3 in Table 4.3. If node misconfiguration were driving the gap, we would expect the result to vary across nodes rather than remain consistent. The second reason is also unlikely given that the GPU nodes, particularly on models one would expect to parallelize computations well, often outperformed the CPU-only nodes in raw latency, with N-HiTS being a clear example (Figure 4.11, Figure 4.12). The biggest residual risk is that the GPU nodes were somewhat artificially slowed by a node misconfiguration, resulting in a comparatively small latency gain rather than being entirely dominated.

The implication for the rest of this work is that the empirical GPU ceilings reported in Table 4.3 should be read as a lower bound on the true hardware capability rather than a tight characterization of it. The distance from a workload point to the ceiling on a GPU roofline is therefore not directly comparable to the same distance on a CPU roofline, since the GPU ceiling itself is likely understated. Figure 4.4 shows the same workload points against the theoretical ceiling, where the gap is substantially wider than against the empirical one. The under-utilization of GPU compute therefore holds under either ceiling, which means the case for provisioning top-tier accelerators on these workloads has to rest on absolute latency rather than on efficient hardware use.

5.2.2 Roofline Bounds and Overall Utilization

When looking at the roofline results, it becomes quite clear that all of the models sit inside even the lowest roofline bound. This result is expected. The reason is that we include everything for an entire runtime inference pass, including data normalization and processing. These operations are impacted by potential sequential bottlenecks, as well as python work such as dispatch overhead. All of these sequential, often bandwidth bound operations both move the models further to the left on the operational intensity dimension, and down on the performance dimensions.

This very same full-model average is also a probable cause for the narrow OI band that we find most of the models in, namely between about 2-7 FLOPs/byte. Even though the models themselves have very different algorithmic OI, at our chosen level of profiling these are compressed into a much narrower band. This means that at the granularity that matters for deployment, the architectural class may be less of a predictor of where a model lands on the roofline than one might initially expect, even though it still has some relevance. A useful sanity check is that this band stays roughly the same across hardware tiers, as noted in section 4.4. This is the expected behavior given that OI is a property of the algorithm rather than the hardware (see subsection 2.2.3), and wildly varying OI for the same model across machines would have suggested issues with the instrumentation itself.

A related observation from section 4.4 is that despite the OI band staying narrow across hardware, the ridge point itself shifts considerably between tiers, which means the same workload can fall on opposite sides of the ridge depending on the machine. On the H100 the ridge sits at roughly 15 FLOP/byte, while on the F8s_v2 it sits at roughly 4 FLOP/byte, and as a result the same models are bandwidth-bound on the H100 while several of them are closer to compute-bound on the CPU. This directly tells us that whether a forecasting workload is bandwidth- or compute-bound on commodity cloud hardware is as much a property of which hardware tier it runs on as of the architecture itself.

The same averaging effect also explains why being on the compute-bound side of the ridge does not translate to approaching peak compute, as noted in section 4.4 for the F8s_v2 and NC40ads_H100_v4 results. The roofline classification reflects which ceiling a kernel is limited by in principle, but at the model level with framework overhead included, both classifications collapse toward well below either ceiling. On the F8s_v2 the fastest models reach a few GFLOP/s against an empirical peak of roughly 370 GFLOP/s, and on the NC40ads_H100_v4 they reach around 10 GFLOP/s against a peak of roughly 1250 GFLOP/s. The gap therefore widens on the H100, and combined with the earlier point that the empirical GPU ceiling is itself is conservative in comparison to the theoretical, the actual deployment headroom on the H100 is likely larger than even these numbers suggest.

However, since we are not focusing on optimizing computational kernels, but rather mapping out the hardware utilization and behavior, this provides us with extra information. It shows us where we are in relation to full-load hardware behavior, which gives a clearer picture of how much of the hardware actually remains unused in normal operation. The roofline therefore provides a utilization signal more

than a kernel-optimization guide, and hints at the true performance headroom that production-grade optimization could close. Worth noting though, is that this is not a full-blown production grade deployment, which means that potential optimizations that can target both the target hardware and model are missing. Such optimizations could affect the results, but given that the base implementations for the models are of similar optimization levels already, these adjustments should apply equally to all the models.

5.2.3 Individual Model Behavior

When looking at the individual model behavior, the most notable is TSMixer, which is substantially under-utilizing the available compute as compared to the other models, sitting roughly two orders of magnitude below the rest. This could be due to the overhead of python dispatches or serialization somehow overly affecting TSMixer, perhaps by the operator mix of the model utilizing a number of very small mixing MLPs that the profiler attributes few FLOPs to, but that still cause python dispatch overhead. This interpretation is also consistent with the analytical nature of the profiler FLOP counts discussed in subsection 3.5.3, where operators with limited or no FLOP formula coverage will contribute fewer FLOPs than the work they actually perform. It could also be that during adaptation a critical bug was imposed that made the data-loading or adaptation increase massively compared to the other models through lessened parallelism. The consistency check between the minimally and maximally adapted versions of each model described in subsection 3.4.2 should have caught a structural bug in the adaptation itself, which makes the profiler-attribution hypothesis the more likely of the two, though it does not fully rule out the alternative.

The other notable outlier is TFT, which most fully utilized the hardware when looking at the FLOPs. This is not unexpected considering that it is based on the transformer architecture, which has substantial parallelism potential. However, the picture is not quite that clean, since TFT is also one of the natively multivariate models that processes the full panel jointly, while several of the lower-utilization models were adapted to a batched per-SKU formulation that, as predicted in subsection 3.4.3, shifts the workload toward memory traffic. The architectural explanation and the data-formulation explanation are therefore partly conflated here, and TSMixer being natively multivariate yet sitting at the bottom of the achieved performance range complicates a simple reading where native multivariate models uniformly land higher on the roofline.

5.3 Scaling

The following section covers the scaling results, and what the implications of it are. Firstly, the results will be analyzed in general, then some model specific outliers will be covered and lastly there will be some discussion on the implications on deployment hardware selection.

5.3.1 General Scaling Behavior

The first discussion point is the simple sanity check of how batch size affected the OI. Since OI is the ratio of work to data movement, and increasing batch typically scales both proportionally, the ratio should stay put. This is what the results show across the tested models, with batch sweeps at fixed context and horizon leaving inference OI unchanged as indicated in section 4.6. The accompanying rise in FLOP/s can be explained by largely fixed per-call overheads, such as python dispatch and other functions, getting amortized across the samples. This confirms that OI captures what it should, and grounds the findings for horizon and context.

Beyond this, increasing the context length shifts the workload toward higher OI for most of the tested models, namely TSMixer, K^2VAE, N-HiTS, and xLSTMTime. A likely explanation is that the per-element parameter cost stays roughly the same, while the per-element compute work grows with the context, increasing the total ratio. This also indicates that the extra overhead of loading more data is overtaken by the additional compute required, for most of these models.

A related observation across the tables is that training OI is consistently higher than inference OI, which is expected since the backward pass roughly doubles the FLOPs while activations need to be stored for backprop. The magnitude of this gap varies across models, which hints at different hardware optima for serving compared to training.

5.3.2 Architectural Behaviors

While the general pattern holds for most of the tested models, three architectures deviate from it in ways that warrant closer examination.

The first is K^2VAE, which is the only model where horizon strongly affects total OI. This is partly explained by the architecture in subsubsection 2.1.7.1, since it runs a Kalman filter with predict and update steps per horizon step. Each step propagates work through learnable transition and observation matrices, adding compute that scales with horizon rather than with input size. The other tested models use multi-horizon projection mechanisms that only affect the final projection layer, which is a lower fraction of total work.

The second deviation concerns TFT, which sits in a notably narrow OI band as context grows. Since it is a transformer model, with the established $O(n^2)$ computational pattern of attention from subsection 2.1.3, the ratio of compute to data should increase rapidly for every added context point, driving up OI compared to the other models. A plausible explanation is that the fixed costs of the TFT variable selection networks and the GRNs are large enough to dominate the compute costs of attention for the swept context sizes. This could also point to the selective attention mechanism working well at our tested sizes, with the scaling not yet kicking in until larger contexts are reached.

The final and most unusual deviation comes from TimeFilter, which behaved differently from the rest of the tested models, with inference OI falling as context grew,

and a large gap between training and inference OI. The most likely cause is that the MoE gating mechanism, as explained in subsubsection 2.1.6.1, activates fewer experts on average at inference, since the graph is built from $C \times N$ nodes and the routing selects a variable subset of edges per node, and the top-p allocation method is used. As context grows, the number of activated experts per node may shrink further, lowering achieved FLOPs while the data movement requirements stay similar, since the full graph and all expert weights must still be loaded into memory.

5.3.3 Implications for Hardware Selection

These scaling findings directly inform the hardware selection question raised in RQ3, by indicating which workload parameters shift the cost-latency answer. For models that follow the dominant pattern, where OI rises with context (TSMixer, K^2VAE, N-HiTS, xLSTMTime), operators running longer-context deployments should weigh compute capacity more heavily, since the workload moves further into the compute-bound region as context grows. For models where inference OI stays flat, such as TFT, bandwidth remains the dominant constraint regardless of context length. The same flat-OI pattern was observed for TimeFilter, but as discussed in subsection 5.3.2, this likely reflects an architectural quirk of the MoE gating rather than a stable deployment property, and should be treated with caution as a basis for hardware selection.

The most actionable finding is K^2VAE, which is the only model where horizon meaningfully shifts the hardware preference. Deployments of models using similar Kalman-filter or recurrent state-update paradigms should weigh the compute requirements of both context and horizon when selecting hardware, rather than treating horizon as a free parameter.

More broadly, these findings are illustrative of how a similar framework can be used for hardware selection rather than universal guidance. Operators deploying a specific architecture on a specific workload should run the framework on their own configuration, since the scaling behavior depends on architectural choices, data characteristics, and the particular hyperparameter regime in use.

5.4 Cost-Latency Efficiency

Looking at CPU instances, we see a clear linear cost increase depending on series and vCPU count. However, this cost increase is not much reflected in latency reduction, and often resulted in an increase rather than a reduction in latency. This could be due to actual model inference being a smaller amount of the total model runtime, which as previously mentioned in subsection 5.2.2 also includes data preparation. Due to their sequential nature, this could add a substantial amount of time that is largely unaffected by the vCPU count, as an effect of Amdahl's law.

Additionally, splitting a relatively small inference task into many different threads could incur overhead from several factors, which could include scheduling, cross-thread communication, and synchronization. This could explain the negative rela-

tionship between core count and model latency.

On GPU instances, we sometimes see a similar relationship where multiple GPUs increase latency rather than reduce it. For these models, there could be similar factors contributing as that of CPU instances. It may be the case that models uses a high volume of cross-GPU communication. Both through GPU interconnects or going through the CPU, there is an incurred transfer overhead. One other possible reason could be that some models are so small that the repeated GPU round-trips from CUDA kernel launches through the PCIe bus takes longer than what it does for a CPU to simply keep running inference tasks.

Even though many models yield GPU performance that clearly beats CPU performance, there are also some architectures which the inverse happens. For these workloads, it is more optimal to use CPU-only instances rather than paying the premium for a GPU node and leaving the accelerator unutilized.

When comparing SKUs for the p50 of all models, it is clear that the NC8as_T4_v3 provides the best middle ground performance between runtime cost and response latency. Due to its older hardware, it becomes considerably cheaper than other GPU-accelerated SKUs but remains very beneficial in accelerating models more than just high-compute CPU SKUs.

This also becomes a large benefit when running models in a production environment. In this report we tested conditions for steadily serving models, but when serving these models to users, bursty traffic may skew cost upwards during idle time. H100 SKUs have a disproportionately higher cost compared to T4 SKUs. This may in reality, due to under-utilization, lead to even higher total runtime costs for serving the same users.

5.5 Synthetic and Real-World Data

When observing forecasting on real data compared to synthetic data, we see a difference in computational characteristics when looking at Figure 4.5. Mainly, there is a performance degradation that can be observed when using real-world ERP data compared to simple, synthetic data.

This trend is much broken by the NC64as_T4_v3 which achieved better performance on real-world data. It may be that models do run better on multi-GPU due to batching behavior of real data, however, this does trend is not followed by the We cannot easily tell if this is the case, due to having a sample size of two SKUs. More multi-GPU instances would be needed to be tested to know if this holds true.

When comparing models themselves, we saw that most models exhibited a higher latency and lower FLOP/s compared to synthetic data, with this behavior inverted by TSMixer and xLSTMTime. For these models, we may get latency spikes that are not as predictable, and could give spikes or “dips” in latency and/or FLOPS performance. In TFT, there was practically no change between real and synthetic data, which points at a robust execution path that remains mostly unchanged no

matter the data shape. Between these models there comes a trade-off between high and low predictability when running them.

6

Conclusion

This thesis asked how the cost-latency trade-offs of deep learning forecasting architectures behave across commodity cloud hardware, and whether a workload-aware benchmarking framework could turn infrastructure decisions into something more grounded than heuristics. The framework built for this purpose, together with the empirical findings it produced, forms the contribution of this work.

The roofline analysis shows that architecture and hardware cannot be characterized in isolation. Across the tested architectures, operational intensity clustered in a narrow band of roughly 2 to 7 FLOPs/byte, which is a tighter range than the five architectural classes would suggest. At the same time, the ridge point shifted from around 4 FLOPs/byte on the lowest-tier CPU instance to roughly 15 FLOPs/byte on the H100, placing the same workload on opposite sides of the ridge depending on the machine. The boundedness of a model is therefore a property of the model-hardware pair, and hardware selection should be guided by the bandwidth and compute pattern of the actual execution rather than by architectural class alone.

Most models increase their OI as context length grows, so the amount of history used for forecasting raises the compute requirement in most deployments. The exception was K^2VAE, where horizon length and context, drove the shift in computational profile. This is consistent with its Kalman-filter design, in which work scales per horizon step, and it suggests that other models built around horizon-driven state updates may inherit the same scaling behavior.

Comparing real and synthetic data, the latency and FLOP/s ratios stayed close to 1.0 for most model and SKU pairs, so synthetic benchmarks reliably approximate real-world behavior for an initial pass. A minority of configurations diverged noticeably, however, which means synthetic data is suitable for narrowing the search space but not for committing to a final deployment.

On the cost-latency frontier, the assumption that larger CPUs or state-of-the-art accelerators yield better serving performance does not hold for the workloads tested. Lightweight models such as TSMixer were served most efficiently on low-cost CPU instances, where vertical scaling increased cost without reducing, and sometimes worsened, latency. For our workloads, the older, lower-tier, single-GPU SKU held the best overall trade-off between cost and latency, and operators can treat this tier as a reasonable starting point before sweeping their own configurations.

Overall, the method used to find these patterns is more durable than the numerical

results, since the dominant factor is the model-hardware pair rather than either side in isolation. The specific numbers are also conditional on methodological choices discussed earlier, including the framework-level profiling boundary and the likely understatement of the empirical GPU ceilings. The methodology itself is intended to generalise, and a similar sweep can be run for any other combination of architectures and instances to guide provisioning. Even lower-tier hardware can be a cost-effective option for the kinds of ERP forecasting workloads used in this study.

6.1 Future Work

Future research should transition from framework-level software estimations to hardware-level profiling. The current methodology relies on analytical FLOPS derivations from the PyTorch profiler. Implementing hardware counters would yield empirical operational intensity measurements, permitting a more accurate comparison between actual workload execution and theoretical hardware rooflines.

Currently, GPU ERT has likely inaccurately measured ceilings. The current toolkit produced compute ceilings orders of magnitude below theoretical limits, indicating a failure to fully saturate the GPU compute units. A more robust profiling methodology is required to create bounds that are as close as possible to theoretical limits for accelerator hardware.

This report has been exploring the pricing of available VMs that were reserved for the duration of our benchmarks. However, due to the previously mentioned sub-100% utilization, evaluating serverless environments could provide a more planable architecture and make for a more realistic model.

As the data set that we were using can be considered small in the context for e.g. transformer models, using a much larger data set could improve not only the ability for models to generalize much better, but also potentially change the computational behavior of some models.

Regarding hardware, more than just x86 CPUs and NVIDIA GPUs could be explored. Architectures such as ARM and RISC-V could yield higher efficiency and lower cost, and more dedicated accelerators such as tensor processing units could be evaluated.

Lastly, future work could cover the two missing models handled herein to see if the general patterns hold, or change. Furthering the understanding of how different architectural classes behave across wide sets of hardware.

Bibliography

[1] P. Global, *Finance Trends CFO Priorities 2025*, 2025. Accessed: Dec. 11, 2025. [Online]. Available: https://www.protiviti.com/gl-en/survey/global-finance-trends-survey.

[2] P. De Rosa, Y.-D. Bromberg, P. Felber, D. Mvondo, and V. Schiavoni, "On the cost of model-serving frameworks: An experimental evaluation," in *2024 IEEE International Conference on Cloud Engineering (IC2E)*, 2024, pp. 221–232. DOI: 10.1109/IC2E61754.2024.00032.

[3] E. Erdil, *Inference economics of language models*, 2025. arXiv: 2506.04645 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2506.04645.

[4] J. Park et al., *Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications*, 2018. arXiv: 1811.09886 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1811.09886.

[5] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 397–411, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/romero.

[6] M. LeMay, S. Li, and T. Guo, "Perseus: Characterizing performance and cost of multi-tenant serving for cnn models," in *2020 IEEE International Conference on Cloud Engineering (IC2E)*, 2020, pp. 66–72. DOI: 10.1109/IC2E48712.2020.00014.

[7] A. Vaswani et al., "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 9781510860964.

[8] R. Jayanth, N. Gupta, and V. Prasanna, "Benchmarking edge ai platforms for high-performance ml inference," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 2024, pp. 1–7. DOI: 10.1109/HPEC62836.2024.10938499.

[9] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzàlez, "E-pur: An energy-efficient processing unit for recurrent neural networks," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 18, ACM, Nov. 2018, pp. 1–12. DOI: 10.1145/3243176.3243184. [Online]. Available: http://dx.doi.org/10.1145/3243176.3243184.

[10] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024. DOI: 10.1109/MM.2024.3373763.

[11] Y. Lu et al., "Machine learning for synthetic data generation: A review," *arXiv preprint arXiv:2302.04062*, 2023.

[12] Lindholm, F. Lindsten, T. B. Schön, and N. Wahlström, *Machine learning: A first course for engineers and scientists*. Cambridge University Press, 2022.

[13] Y.-C. Chou, H. H.-C. Chuang, P. Chou, and R. Oliva, "Supervised machine learning for theory building and testing: Opportunities in operations management," *Journal of Operations Management*, vol. 69, no. 4, pp. 643–675, Jan. 2023. DOI: 10.1002/joom.1228.

[14] B. Rolf et al., "A review on unsupervised learning algorithms and applications in supply chain management," *International Journal of Production Research*, vol. 63, no. 5, pp. 1933–1983, Aug. 2024. DOI: 10.1080/00207543.2024.2390968.

[15] H. Chau, D. Nguyen, and T. Nguyen, "Continuous-time optimal investment with portfolio constraints: A reinforcement learning approach," *European Journal of Operational Research*, vol. 328, no. 3, pp. 1068–1092, Feb. 2026. DOI: 10.1016/j.ejor.2025.08.032.

[16] R. P. Masini, M. C. Medeiros, and E. F. Mendes, "Machine learning advances for time series forecasting," *Journal of Economic Surveys*, vol. 37, no. 1, pp. 76–111, Jul. 2021. DOI: 10.1111/joes.12429.

[17] J. A. Mariño, M. E. Arrieta-Prieto, and S. A. Calderón V., "Comparison between statistical models and machine learning for forecasting multivariate time series: An empirical approach," *Communications in Statistics: Case Studies, Data Analysis and Applications*, vol. 11, no. 1, pp. 56–91, Jan. 2025. DOI: 10.1080/23737484.2025.2463905.

[18] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd. USA: Prentice Hall PTR, 1998, ISBN: 0132733501.

[19] K. Gurney, *An Introduction to Neural Networks*. USA: Taylor & Francis, Inc., 1997, ISBN: 1857286731.

[20] B. N. Oreshkin, D. Carpov, N. Chapados, and Y. Bengio, *N-beats: Neural basis expansion analysis for interpretable time series forecasting*, 2020. arXiv: 1905.10437 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1905.10437.

[21] C. Challu, K. G. Olivares, B. N. Oreshkin, F. Garza, M. Mergenthaler-Canseco, and A. Dubrawski, *N-HiTS: Neural hierarchical interpolation for time series forecasting*, 2022. arXiv: 2201.12886 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2201.12886.

[22] S.-A. Chen, C.-L. Li, N. Yoder, S. O. Arik, and T. Pfister, *Tsmixer: An all-mlp architecture for time series forecasting*, 2023. arXiv: 2303.06053 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2303.06053.

[23] F. M. Bianchi, E. Maiorino, M. C. Kampffmeyer, A. Rizzi, and R. Jenssen, *Recurrent neural networks for short-term Load forecasting: An overview and comparative analysis*. Springer, 2017.

[24] L. Johnston, V. Patel, Y. Cui, and P. Balaprakash, "Revisiting the problem of learning long-term dependencies in recurrent neural networks," *Neural Networks*, vol. 183, p. 106 887, Mar. 2025. DOI: 10.1016/j.neunet.2024.106887.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[26] G. Van Houdt, C. Mosquera, and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5929–5955, May 2020. DOI: 10.1007/s10462-020-09838-1.

[27] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, 1999, 850–855 vol.2. DOI: 10.1049/cp:19991218.

[28] M. Alharthi and A. Mahmood, *Xlstmtime : Long-term time series forecasting with xlstm*, 2024. arXiv: 2407.10240 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2407.10240.

[29] B. Lim, S. Ö. Ark, N. Loeff, and T. Pfister, "Temporal fusion transformers for interpretable multi-horizon time series forecasting," *International journal of forecasting*, vol. 37, no. 4, pp. 1748–1764, 2021.

[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[31] L. Chen, S. Li, Q. Bai, J. Yang, S. Jiang, and Y. Miao, "Review of image classification algorithms based on convolutional neural networks," *Remote Sensing*, vol. 13, no. 22, 2021, ISSN: 2072-4292. DOI: 10.3390/rs13224712. [Online]. Available: https://www.mdpi.com/2072-4292/13/22/4712.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[33] H. Wu, T. Hu, Y. Liu, H. Zhou, J. Wang, and M. Long, "Timesnet: Temporal 2d-variation modeling for general time series analysis," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=ju_Uqw384Oq.

[34] S. Masoudnia and R. Ebrahimpour, "Mixture of experts: A literature survey," *Artificial Intelligence Review*, vol. 42, no. 2, pp. 275–293, 2014. DOI: 10.1007/s10462-012-9338-y.

[35] G. Woo, C. Liu, A. Kumar, C. Xiong, S. Savarese, and D. Sahoo, *Unified training of universal time series forecasting transformers*, 2024. arXiv: 2402.02592 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2402.02592.

[36] X. Liu et al., "Moirai-MoE: Empowering time series foundation models with sparse mixture of experts," *arXiv preprint arXiv:2410.10469*, 2024. DOI: 10.48550/arXiv.2410.10469.

[37] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[38] Y. Hu et al., *Timefilter: Patch-specific spatial-temporal graph filtration for time series forecasting*, 2025. arXiv: 2501.13041 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2501.13041.

[39] J. Walker, C. Doersch, A. Gupta, and M. Hebert, "An uncertain future: Forecasting from static images using variational autoencoders," vol. 9911, Oct. 2016, pp. 835–851, ISBN: 978-3-319-46477-0. DOI: 10.1007/978-3-319-46478-7_51.

[40] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, 2022. arXiv: 1312.6114 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1312.6114.

[41] C. Doersch, *Tutorial on variational autoencoders*, 2021. arXiv: 1606.05908 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1606.05908.

[42] K. Sohn, H. Lee, and X. Yan, "Learning structured output representation using deep conditional generative models," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28, Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf.

[43] X. Wu, X. Qiu, H. Gao, J. Hu, B. Yang, and C. Guo, "K2vae: A koopman-kalman enhanced variational autoencoder for probabilistic time series forecasting," in *Proceedings of the 42nd International Conference on Machine Learning*, ser. ICML'25, Vancouver, Canada: JMLR.org, 2025.

[44] J. Chen et al., "Run, don't walk: Chasing higher flops for faster neural networks," in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 12 021–12 031. DOI: 10.1109/CVPR52729.2023.01157.

[45] J. Domke et al., "Double-precision fpus in high-performance computing: An embarrassment of riches?" In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 78–88. DOI: 10.1109/IPDPS.2019.00019.

[46] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, "A bf16 fma is all you need for dnn training," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1302–1314, 2022. DOI: 10.1109/TETC.2022.3187770.

[47] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th. Baltimore, MD: Johns Hopkins University Press, 2013.

[48] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, ISSN: 0163-5964. DOI: 10.1145/216585.216588. [Online]. Available: https://doi.org/10.1145/216585.216588.

[49] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024. DOI: 10.1109/MM.2024.3373763.

[50] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[51] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A quantitative approach*, 5th Edition. Morgan Kaufmann Publishers, 2011.

[52] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000, ISSN: 0360-0300. DOI: 10.1145/358923.358939. [Online]. Available: https://doi.org/10.1145/358923.358939.

[53] P. J. Denning, "Working set analytics," *ACM Comput. Surv.*, vol. 53, no. 6, Feb. 2021, ISSN: 0360-0300. DOI: 10.1145/3399709. [Online]. Available: https://doi.org/10.1145/3399709.

[54] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[55] B. Gregg, *Systems performance: Enterprise and the cloud*. Prentice Hall, 2014.

[56] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. [Online]. Available: https://doi.org/10.1145/2408776.2408794.

[57] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17, Boston, MA, USA: USENIX Association, 2017, pp. 613–627, ISBN: 9781931971379.

[58] K. Miettinen, *Nonlinear multiobjective optimization*. Springer, 2012.

[59] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, 2002. DOI: 10.1109/MM.2002.1028477.

[60] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, "Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 284–295, ISBN: 9781450360791. DOI: 10.1145/3330345.3330371. [Online]. Available: https://doi.org/10.1145/3330345.3330371.

[61] AMD. "AMD uPROF," Accessed: Apr. 29, 2026. [Online]. Available: https://www.amd.com/en/developer/uprof.html.

[62] Intel. "Intel Advisor," Accessed: Apr. 29, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html.

[63] NVIDIA. "NVIDIA Nsight Compute," Accessed: Apr. 29, 2026. [Online]. Available: https://developer.nvidia.com/nsight-compute.

[64] NVIDIA. "NVIDIA Nsight Systems," Accessed: Apr. 29, 2026. [Online]. Available: https://developer.nvidia.com/nsight-systems.

[65] Friedrich-Alexander University, Erlangen National High-Performance Computing Center. "LIKWID," Accessed: Apr. 29, 2026. [Online]. Available: https://hpc.fau.de/research/tools/likwid.

[66] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000, ISSN: 1094-3420. DOI: 10.1177/109434200001400303. [Online]. Available: https://doi.org/10.1177/109434200001400303.

[67] T. Röhl, J. Treibig, G. Hager, and G. Wellein, "Overhead analysis of performance counter measurements," in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 176–185. DOI: 10.1109/ICPPW.2014.34.

[68] PyTorch Team. "torch.profiler - PyTorch 2.11 documentation," Accessed: May 4, 2026. [Online]. Available: https://docs.pytorch.org/docs/2.11/profiler.html.

[69] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, "Xsp: Across-stack profiling and analysis of machine learning models on gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 326–327. DOI: 10.1109/IPDPS47924.2020.00042.

[70] D. Narayanan et al., "Efficient large-scale language model training on gpu clusters using megatron-lm," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[71] Q. Zhao et al., *Deepcontext: A context-aware, cross-platform, and cross-framework tool for performance profiling and analysis of deep learning workloads*, 2024. arXiv: 2411.02797 [cs.PF]. [Online]. Available: https://arxiv.org/abs/2411.02797.

[72] Lawrence Berkeley National Laboratory, *CS roofline toolkit*. Accessed: May 21, 2026. [Online]. Available: https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/.

[73] SchedMD, *Slurm workload manager - overview*. Accessed: May 25, 2026. [Online]. Available: https://slurm.schedmd.com/overview.html.

[74] EnvModules, *Environment modules*. Accessed: May 25, 2026. [Online]. Available: https://envmodules.io/.

[75] X. Lou, K. Chen, G. Xu, H. Qiu, S. Guo, and T. Zhang, "Protecting confidential virtual machines from hardware performance counter side channels," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 195–208. DOI: 10.1109/DSN58291.2024.00031.

A

Model Choices

A.1 All Models Considered

A.2 Open-Source Models Scored and Chosen

Name	Category	Composition	Score	Repo	Year	Article	Description	Performance characteristics	Notes
Temporal fusion transformer	Transformer		5	https://github	2025	https://arxiv.org/abs/2511.00552	Probabilistic forecasting, used for explainable predictions	O(L^2)	Alternative source, none linked in paper. Very relevant architecture. However, the paper has a short (5w) horizon. Must be adapted if used.
MultiPatchFormer	Transformer		5	https://github	2025	https://www.nature.com/articles/s41598-024-82417-4			May not handle just monthly data predictions well, but weekly and daily. May also have issues with adapting to new SKUs with limited data.
TimeXer	Transformer		5	https://github	2024	https://arxiv.org/abs/2402.19072	Like PatchTST but with improved cross-variable performance?		Interesting if you have many data points, and accessible external variables. Good architecture for what we want.
Moirai	Transformer		5	https://github	2024	https://arxiv.org/pdf/2402.02592			Seems useful and adaptable for our use case.
K2VAE	Other/Generic		5	https://github	2025	https://arxiv.org/html/2505.23017v1			Seems very fitting for our use-case, however, it would require a lot of learning to adapt to work for our use case.
N-HiTS	Other/Generic		5	https://github	2022	https://arxiv.org/abs/2201.12886	Same as N-BEATS, but has different sampling rates in the blocks to utilize resolution. Reduces parameters and increases accuracy		Strong candidate, but may have issues handling external variables without significant rearchitecting.
TSMixer	MLP		5	https://github	2023	https://arxiv.org/pdf/2303.06053.pdf			Multivariate, good. Should use extended variant instead.
SAMoVAR	Hybrid	Transformer + VAR	5	https://github	2025	https://arxiv.org/abs/2502.07244	Combines models for improved performance, interpretability, and computational efficiency	O(L) (allegedly)	Could be pretty relevant, especially if we have lots of daily data that we could aggregate. Could fail on too many SKUs as it creates a 2D matrix, but if it's a lower number (hundreds), it may still work well.
Moirai-MoE	Hybrid	Transformer + MoE	5	https://github	2024	https://arxiv.org/abs/2410.10469	Eliminates frequency group buckets by using sparse MoE in the transformer blocks		Interesting and with patterns that match what we want. Would require some rearchitecting.
Autoformer	Hybrid	Transformer + Auto-Correlation	5	https://github	2021	https://arxiv.org/abs/2106.13008	Aimed for LTSC		Good candiate, may have issues with needing lots of data to train.
ModernTCN	CNN		5	https://github	2024	https://proceedings.iclr.cc/paper_files/paper/2024/file/86b1437c1e4c3b3c4debff98234a67e7-Paper-Conference.pdf	Improve receptive field by increasing convolution kernel sizes in the convolutional layers.		This is very relevant and would probably be somewhat easy to implement. The largest potential issues is with sparsity and the mixed frequency.
TimesNet	CNN		5	https://github	2023	https://arxiv.org/pdf/2210.02186	CNN model designed for time-series data by making the rows/columns based on the intra/inter period data.		Likely to work well, and easy to adapt. Worthy candidate.
iTransformer	Transformer		4	https://github	2023	https://arxiv.org/abs/2310.06625v4	Good at handling how variables affect each other, tokenizes the variables rather than time		Highly interesting. Requires less GPU resources and trains faster.
Crossformer	Transformer		4	https://github	2023	https://openreview.net/pdf?id=vSVLM2j9eie	Multivariate		Would require a lot of data, and then it might also struggle with some covariates that we manually tell it based on author's test.
TimeFilter	Other/Generic	GNN	4	https://github	2025	https://arxiv.org/pdf/2501.13041			Interesting and could support multiple various input, but might require a bit of rearchitecting to fit our data well.
TimeMixer	MLP		4	https://github	2024	https://proceedings.iclr.cc/paper_files/paper/2024/file/a7ac8a21e5a27e7ab31a5f42a0117bdb-Paper-Conference.pdf	Combines larger and smaller predictions (daily, weekly, monthly) to smooth out large varianes and increase performance		Good architecture, that mostly matches our use case.
xLSTM	LSTM		4	https://github	2024	https://arxiv.org/pdf/2407.10240	Incorporates a lot of learnings from transformers and apply them in LSTM contexts	O(L)	Strong candidate that should be relatively easy to rearchitetct for our purposes.
Time-MoE	Hybrid	Transformer + MoE	4	https://github	2024	https://arxiv.org/pdf/2409.16040			Multivariate, multi-resolution. Could be problematic as it breaks out multivariate into multiple univariate
Pyraformer	Transformer		3	https://github	2022	https://openreview.net/pdf?id=0EXmFzUn5I			Can process data at different scales simultaneously, can be used to summarize high-res data. Works with covariates. Custom CUDA kernel.
FEDFormer	Transformer		3	https://github	2022	https://proceedings.mlr.press/v162/zhou22g/zhou22g.pdf			Multivariate, but needs data points
SegRNN	RNN		3	https://github	2023	https://arxiv.org/abs/2308.11200	Use segments instead of individual data points to train/make predictions		Potentially overkill, but very relevant. Also tested on continuous data, whereas ours is not.
FreTS	Other/Generic		3	https://github	2023	https://arxiv.org/pdf/2311.06184	Performs forecasting in frequency domain		Would probably work well for trend data, may work less well on lumpy data.
WPMixer	MLP		3	https://github	2024	https://arxiv.org/abs/2412.17176			This is a good contender, but would require a lot of adaptation.
LightTS	MLP		3	https://github	2022	https://arxiv.org/abs/2207.01186			Multivariate, but is bad at handling lumpy data. Continuous preferred
Informer	Hybrid	Transformer + ProbSparse	3	https://github	2025	https://arxiv.org/html/2504.13522v2			Source is alternative to what was linked in the paper. Primarily focuses on stock prediction, but also uses real-world parameters that weigh in. Could be adapted
MSGNet	Hybrid	Transformer + ???	3	https://github	2024	https://dl.acm.org/doi/10.1609/aaai.v38i10.28991			Good candidate that should be good for long time series, but may be bad at adapting to adding new SKUs.
MICN	CNN		3	https://github	2023	https://openreview.net/pdf?id=zt53IDUR1U	Uses dual convolution approach		Could be relevant, but lacks explicit support for external variables. May also struggle with Licence type streams.
SCINet	CNN		3	https://github	2022	https://arxiv.org/pdf/2106.09305	Tries to automatically downsample larger feature spaces.		Not good for very sparse data.
Sundial	Transformer		2	https://github	2025	https://arxiv.org/abs/2502.00816			Seems good, but univariate.
PatchTST	Transformer		2	https://github	2022	https://arxiv.org/abs/2211.14730	Makes patches of the stride to group data together into longer sequences	O(N^2) for the number of patches, not points	Fits very well for our use case. Might be problematic with data granularity. Univariate.
Non-stationary Transformer	Transformer		2	https://github	2022	https://openreview.net/pdf?id=ucNDIDRNjjv			Bad at sparse data. Seemingly difficult to adapt.
Reformer	Transformer		2	https://github	2020	https://openreview.net/forum?id=rkgNKkHtvB			Most likely not enough data points to make this relevant, as the trade-offs are not worth it for our horizon. Additionally, focuses on univariate prediction.
Plain Transfomer	Transformer		2	https://github	2017	https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf			Good principles, but may not be super relevant as more specialized models have been developed based on these principles.
RWKV	RNN		2	https://github	2024	https://arxiv.org/html/2401.09093v1	RNN architecture that take some of the attention learnings from Transformers		Good architecture, but used in the paper for more daily predictions
TSPulse	Other/Generic		2	https://openr	2025	https://arxiv.org/html/2505.13033v1	Pre-trained, ultra compact. "Pre-train once, forecast anywhere"		While cool and efficient, it is primarily focused on analysis and short-term forecasting. They explicitly recommend using something else for long-term forecasting.
MAMBA	Other/Generic		2	https://github	2024	https://openreview.net/pdf?id=tEYskw1VY2	Model developed to explicitly adress the computational shortcomings of transformers.	O(N)	Might not be a great fit as it could be overkill, but otherwise it would be an interesting candidate.
TiDE	Other/Generic		2	https://github	2023	https://arxiv.org/abs/2304.08424	Long-term forecaster		Good candidate. Univariate and dense, but fast to train and can likely handle our data type
FiLM	Other/Generic		2	https://github	2022	https://openreview.net/pdf?id=zTQdHSQUQWc	Made for LTSF		Good fit, but made to smooth out large spikes, and therefore might require us to fail at handling lumpy license data.
N-BEATS	Other/Generic		2	https://github	2019	https://arxiv.org/abs/1905.10437	Can be configured in generic or interpretable modes. Uses blocks that "backcast" and forecast.		Univariate bad, but the model is explainable. Trained on monthly and other granuarities. Good candidate.
P-sLSTM	LSTM		2	https://github	2025	https://ojs.aaai.org/index.php/AAAI/article/view/33303/35458	LSTM utilizing patching to work on different timescales	O(L/P)	Very relevant. However, it does not really think about inter-SKU relationships, and may not handle very noisy data well.
QuantGAN	GAN		2	https://github	2019	https://arxiv.org/abs/1907.06673			Treats variations as statistical noise. Relies on stationarity.
FinCast	Transformer		1	https://github	2025	https://arxiv.org/html/2508.19609v1	Seems focused on financial/stock data rather than company expenditure/income. May not be that relevant		Not super relevant. Perhaps the architecture could be borrowed and adapted, but it would likely require large changes and adaptations.
Chronos	Transformer		1	https://github	2024	https://arxiv.org/abs/2403.07815	Pre-trained model developed by Amazon.		This is a pre-trained model, probably not something we want to use.
TimesFM	Transformer		1	https://github	2024	https://research.google/blog/a-decoder-only-foundation-model-for-time-series-forecasting/	Pre-trained model developed by Google.		Pre-trained, likely not something we want to use.
TimeGPT	Transformer		1	https://github	2023	https://arxiv.org/abs/2310.03589	Pre-trained model developed by Nixtla.		Pre-trained, likely not something we want to use.
ETSFormer	Transformer		1	https://github	2022	https://arxiv.org/abs/2202.01381			Univariate, not good at lumpy data. Meh...
Supply Chains	Other/Generic	GNN	1	https://github	2025	https://cs.stanford.edu/people/jure/pubs/supplychain-gnn-aaai25.pdf	Model with understanding of production functions		Does not seem super relevant, as production information/cost etc is not relevant for software. Perhaps be adapted if we have some kind of data on how much "compute" things use, but otherwise pretty useless.
DLinear	Other/Generic		1	https://github	2023	https://ojs.aaai.org/index.php/AAAI/article/view/26317	Decomposes into trend and seasonal statistics.		Classic shallow NN to compare to, but is possibly too simple for benchmarking.
KalmanNet	Other/Generic		1	https://github	2021	https://arxiv.org/abs/2107.10043			Would require a lot of effort, and is probably not that well suited.
TiRex	LSTM		1	https://github	2025	https://arxiv.org/pdf/2505.23719			Seems very reasonable, but is univariate.
SRLinear	Hybrid	SR + Some linear predictor	1	https://github	2025	https://ieeexplore.ieee.org/document/11007288	Achieved SOTA in May 2025. Lightweight and interpretable	O(g*P*T) for SR searches	Could probably work very well, but would lack some capability of inter-product relations, as you would run it on each SKU individually.
Lag-Llama	Hybrid	Transformer	1	https://github	2023	https://arxiv.org/abs/2310.08278	Pretrained probabilistic model. Gives prediction confidences		Pre-trained, likely not something we want to use.
FinGPT	Transformer		0	https://github	2023	https://arxiv.org/pdf/2306.06031			Very irrelevant. This is more for extracting data than extrapolating.
TabPFN	Transformer		0	https://github	2022	https://arxiv.org/abs/2207.01848			Uses a hammer for a screw. Will not work, completely useless.
Koopa	Other/Generic		0	https://github	2023	https://arxiv.org/pdf/2305.18803			This is not a good fit and would probably require a lot of effort to get working, even if some concepts could be relevant.
Time-LLM	Hybrid	Transformer + "Reprogramming"	0	https://github	2025	https://arxiv.org/html/2505.10213v1	Reprograms an ordinary LLM so that tokens represent time series data, turning predictions into word completion tasks		This is just an LLM where you prompt engineer it to predict the future. This does not seem very viable for our work.
PromptCast	Hybrid	Transformer + uhhhh	0	https://github	2022	https://arxiv.org/abs/2210.08964	Feeds data as sentences to an LLM. Can leverage domain knowledge		This seems to be kind of terrible? LLM based slop.
MarketGAN	GAN		0	https://github	2023	https://arxiv.org/abs/2309.07708	Seems to be for generating synthetic data?		Do not use. This is more for creating synthetic data.

B

Hyperparameter Performance Characteristics

B.1 K2VAE

Batch	Ctx	Hor	Train OI	Train GF/s	Infer OI	Infer GF/s
128	24	3	6.31	23.85	1.71	0.58
128	20	3	6.22	26.88	1.63	0.64
128	24	3	6.31	25.83	1.71	0.68
128	24	6	10.90	34.28	2.24	0.93
128	24	6	10.90	33.77	2.24	0.93
512	24	3	6.31	25.60	1.71	0.67
128	24	3	6.31	25.89	1.71	0.57
16	20	6	4.45	18.57	2.19	0.83
512	20	3	6.22	26.15	1.63	0.62
512	24	3	6.31	25.53	1.71	0.59
128	20	3	6.22	27.24	1.63	0.62
256	16	18	22.47	40.78	2.85	1.25
512	24	3	6.31	24.93	1.71	0.57
128	20	3	6.22	27.69	1.63	0.60
128	24	6	10.90	34.26	2.24	0.89
256	16	24	25.32	36.87	2.92	1.41
256	12	12	16.30	41.30	2.40	0.93
64	16	12	15.71	38.40	2.44	1.04
256	12	18	23.97	43.29	2.83	1.33
16	20	6	4.45	18.54	2.19	0.83
16	20	6	4.45	19.40	2.19	0.94
32	12	24	27.31	41.41	2.91	1.40
256	8	18	25.33	44.74	2.79	1.27
64	8	24	29.31	38.66	2.88	1.35
256	8	18	25.33	45.38	2.79	1.16

B.2 N-HiTS

Batch	Ctx	Hor	Train OI	Train GF/s	Infer OI	Infer GF/s
32	25	3	0.00	3.34	7.44	0.83
32	14	3	0.00	8.38	7.00	0.70
32	26	6	0.00	10.10	7.52	0.80
32	25	3	0.00	9.98	7.44	0.70
32	15	3	0.00	8.75	7.04	0.69
32	14	3	0.00	8.38	7.00	0.62
16	9	6	0.00	4.34	6.83	0.63
64	20	3	0.01	18.94	7.25	0.72
32	23	3	0.00	9.22	7.37	0.78
32	18	3	0.00	8.50	7.17	0.73
32	21	3	0.00	9.35	7.29	0.73
32	33	3	0.00	10.37	7.71	0.91
32	27	6	0.00	9.82	7.55	0.82
128	25	6	0.02	28.52	7.49	0.83
16	7	12	0.00	4.30	6.88	0.66
256	12	18	0.05	53.12	7.20	0.75
64	17	12	0.01	19.32	7.29	0.78
128	29	6	0.02	30.14	7.62	0.88
256	21	12	0.04	57.04	7.43	0.82
16	8	12	0.00	4.68	6.93	0.65
32	29	6	0.00	10.12	7.62	0.84
512	36	24	0.12	83.66	7.98	1.00
32	29	12	0.01	11.42	7.69	0.89
512	36	24	0.12	92.81	7.98	1.00
16	25	18	0.00	6.30	7.64	0.91

B.3 TFT

Batch	Hor	Train OI	Train GF/s	Infer OI	Infer GF/s
16	3	0.01	3.24	3.56	2.82
64	24	0.21	29.39	3.66	8.50
64	24	0.24	32.94	3.66	9.71
64	24	0.21	30.41	3.66	7.87
64	24	0.24	30.15	3.66	9.65
16	3	0.02	4.51	3.59	3.53
64	24	0.21	30.94	3.66	8.61
64	24	0.18	28.11	3.67	6.90
32	3	0.02	6.05	3.56	3.14
64	3	0.06	19.08	3.59	4.34
16	3	0.01	3.35	3.56	2.83
64	12	0.13	17.99	3.63	5.82
16	18	0.04	7.11	3.65	6.02
16	18	0.05	7.44	3.65	5.81
512	24	1.47	65.58	3.67	7.53
256	12	0.60	59.47	3.64	8.66
128	3	0.12	32.60	3.59	4.41
256	24	1.08	60.96	3.66	10.61
128	18	0.35	49.19	3.65	8.79
512	24	1.93	66.04	3.66	9.90
16	6	0.03	7.68	3.63	6.22
64	24	0.33	39.07	3.65	13.13
32	6	0.08	12.84	3.63	9.51
128	6	0.26	49.19	3.63	9.31
64	12	0.19	22.26	3.64	8.28

B.4 TimeFilter

Batch	Ctx	Hor	Train OI	Train GF/s	Infer OI	Infer GF/s
32	10	3	35.00	29.80	1.32	2.94
32	8	3	35.00	30.35	1.32	2.99
32	4	3	35.00	30.14	1.32	2.75
32	4	3	35.00	30.68	1.32	3.25
32	4	3	35.00	30.08	1.32	3.38
32	12	3	35.00	30.14	1.32	3.12
32	16	3	54.21	33.87	1.23	4.39
32	8	3	19.17	28.82	1.46	1.74
512	10	3	55.78	35.84	1.32	3.07
32	4	12	42.23	30.56	1.32	3.38
32	10	12	42.23	29.85	1.32	3.13
32	8	12	42.23	30.48	1.32	3.39
16	14	18	24.53	23.78	1.33	2.71
32	14	18	49.07	29.24	1.33	2.69
128	28	3	142.07	30.59	1.11	8.58
512	36	6	170.77	30.36	1.06	7.96
512	38	6	170.77	30.24	1.06	9.19
256	38	6	170.77	30.90	1.06	8.68
64	24	24	120.19	32.64	1.14	7.45
32	48	3	200.48	30.38	1.02	9.72
256	28	24	137.96	27.07	1.11	6.49
128	36	24	166.43	28.47	1.07	9.05
64	22	3	-	-	-	-
16	18	3	-	-	-	-
32	20	3	-	-	-	-

B.5 TSMixer

Batch	Ctx	Hor	Train OI	Train GF/s	Infer OI	Infer GF/s
32	12	6	3.75	15.20	2.10	0.03
32	12	6	3.75	16.03	2.10	0.03
32	12	6	3.75	14.48	2.10	0.03
32	12	6	3.75	15.03	2.10	0.03
32	12	6	3.75	15.06	2.10	0.03
32	12	6	3.75	15.13	2.10	0.03
32	12	6	3.75	15.32	2.10	0.03
256	18	6	9.69	31.55	2.65	0.04
128	18	6	9.69	32.79	2.65	0.04
32	18	6	5.17	20.93	2.65	0.04
32	18	6	5.17	20.50	2.65	0.05
32	18	6	5.17	20.92	2.65	0.04
32	12	3	3.56	14.92	2.10	0.03
32	24	6	6.37	24.24	3.04	0.06
512	30	3	13.86	57.91	3.33	0.07
32	30	6	7.41	32.47	3.34	0.06
16	24	18	4.02	16.40	3.07	0.05
64	42	12	17.10	45.97	3.78	0.10
64	36	12	15.63	53.70	3.59	0.09
256	24	18	12.07	36.27	3.07	0.06
16	30	18	4.67	17.54	3.37	0.06
128	42	18	17.13	59.05	3.80	0.09
512	12	24	7.21	13.86	2.12	0.03
256	48	18	18.42	58.55	3.96	0.10
256	24	24	12.13	29.90	3.08	0.06

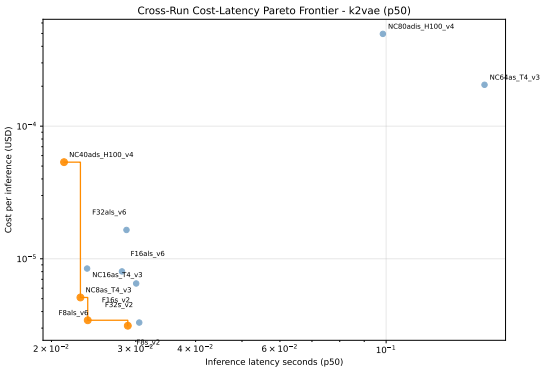
B.6 xLSTMTime

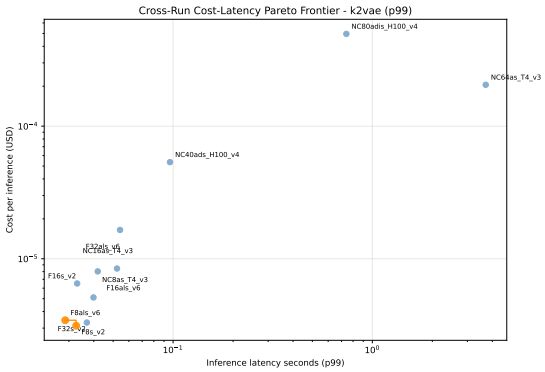
Batch	Ctx	Hor	Train OI	Train GF/s	Infer OI	Infer GF/s
32	12	3	0.06	6.81	1.39	3.34
32	12	3	0.06	6.67	1.39	3.62
32	12	3	0.06	6.74	1.39	3.48
32	18	3	0.09	9.40	1.41	4.40
32	18	3	0.09	9.71	1.41	5.31
32	24	3	0.12	11.68	1.43	7.08
128	18	3	0.36	20.42	1.41	5.14
32	18	3	0.09	9.32	1.41	4.51
32	12	6	0.06	6.75	1.39	3.59
32	18	3	0.09	9.44	1.41	4.17
32	18	3	0.09	9.44	1.41	5.36
32	12	6	0.06	6.76	1.39	3.57
32	12	6	0.06	6.20	1.39	3.04
512	30	3	2.60	35.92	1.45	8.04
32	30	3	0.16	13.77	1.45	9.01
256	18	6	0.76	24.70	1.41	4.46
16	24	18	0.09	7.62	1.43	5.77
256	24	18	1.37	27.53	1.43	5.71
16	30	18	0.11	8.79	1.45	6.52
512	12	24	1.47	21.70	1.39	3.32
256	24	24	1.61	28.94	1.43	6.71
64	36	12	0.49	24.16	1.47	10.73
256	48	18	3.20	36.68	1.50	14.88
64	42	12	0.59	27.22	1.49	11.02
128	42	18	1.35	31.26	1.49	12.52

C

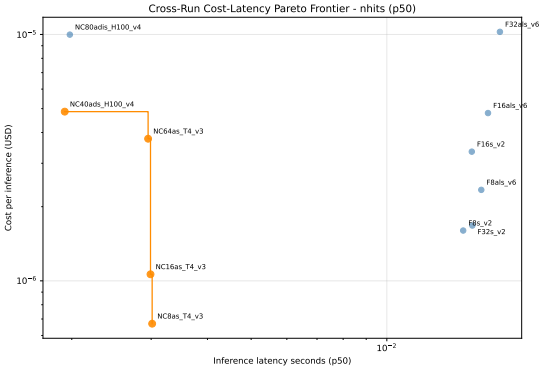
Per-Model Cost-Latency Pareto Frontiers

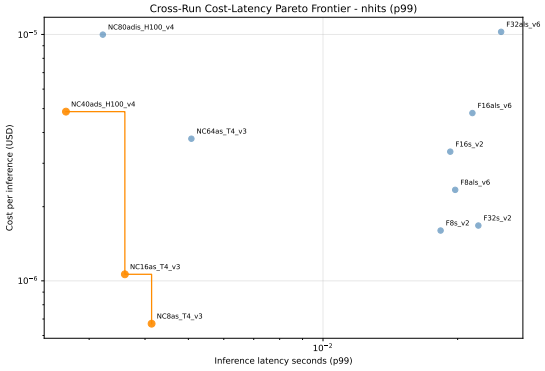
C.1 K2VAE



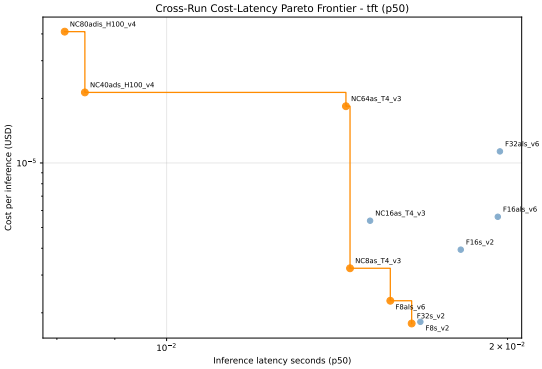


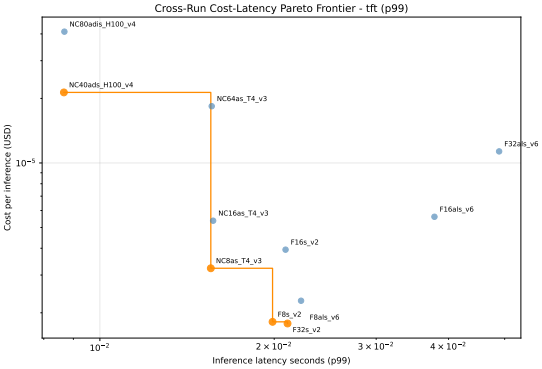
C.2 N-HiTS



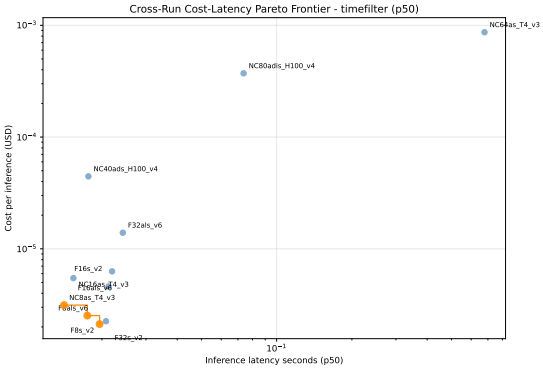


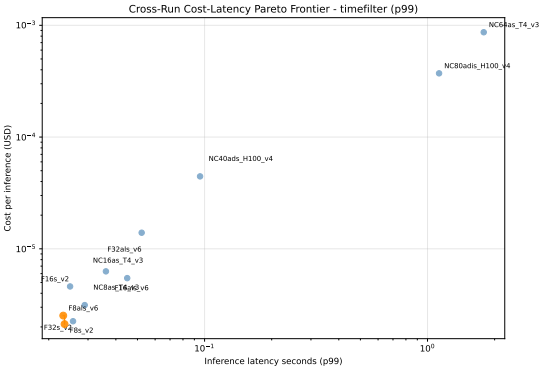
C.3 TFT



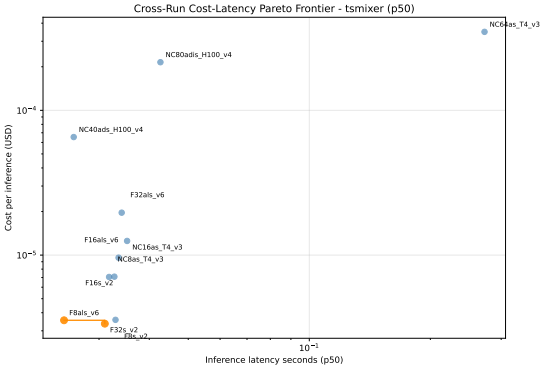


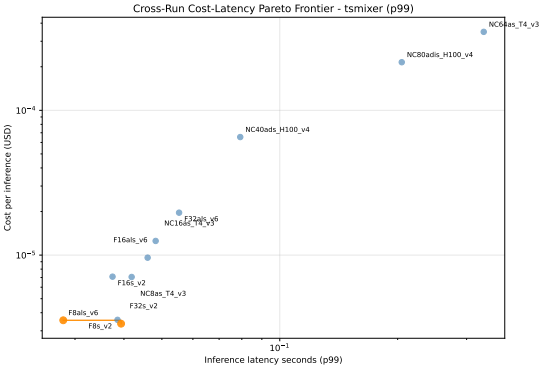
C.4 TimeFilter



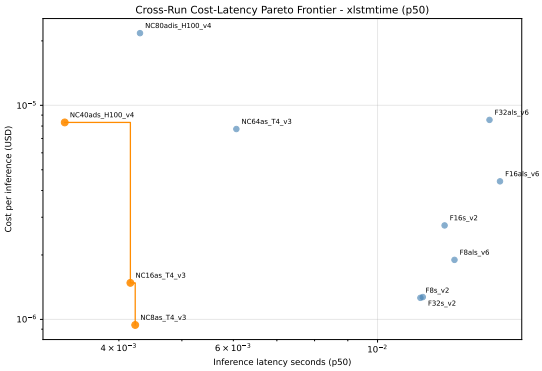


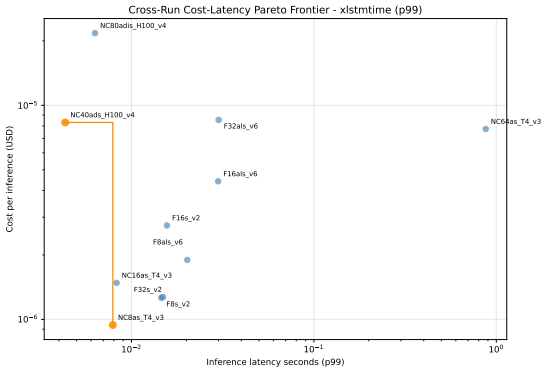
C.5 TSMixer





C.6 xLSTMTime



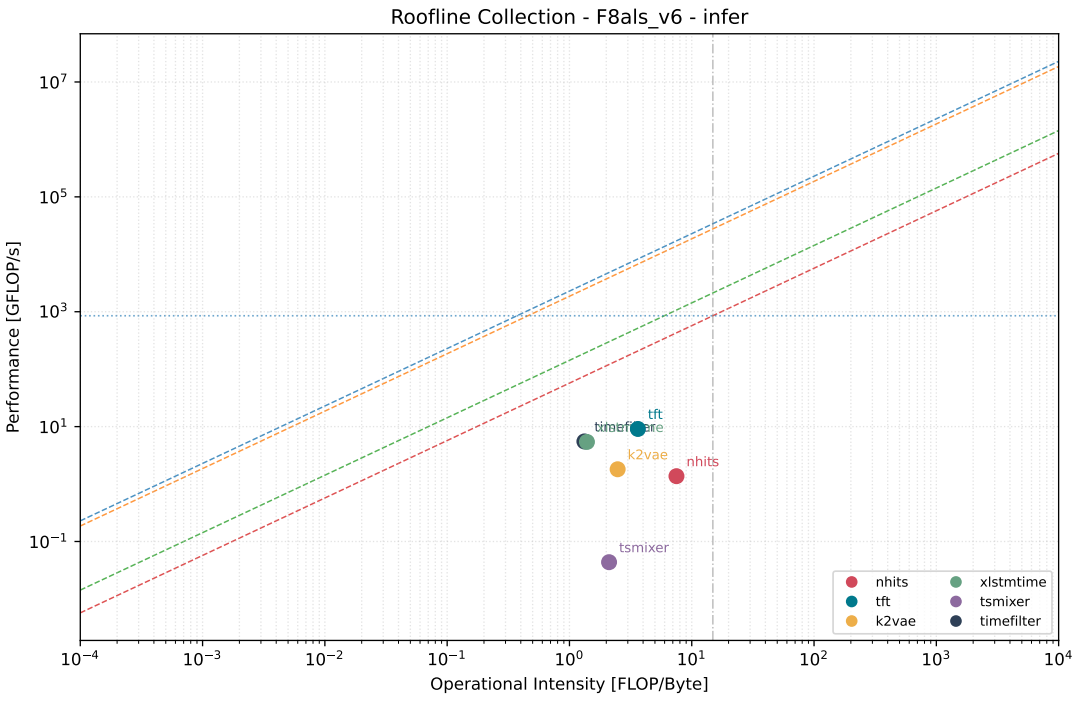


D

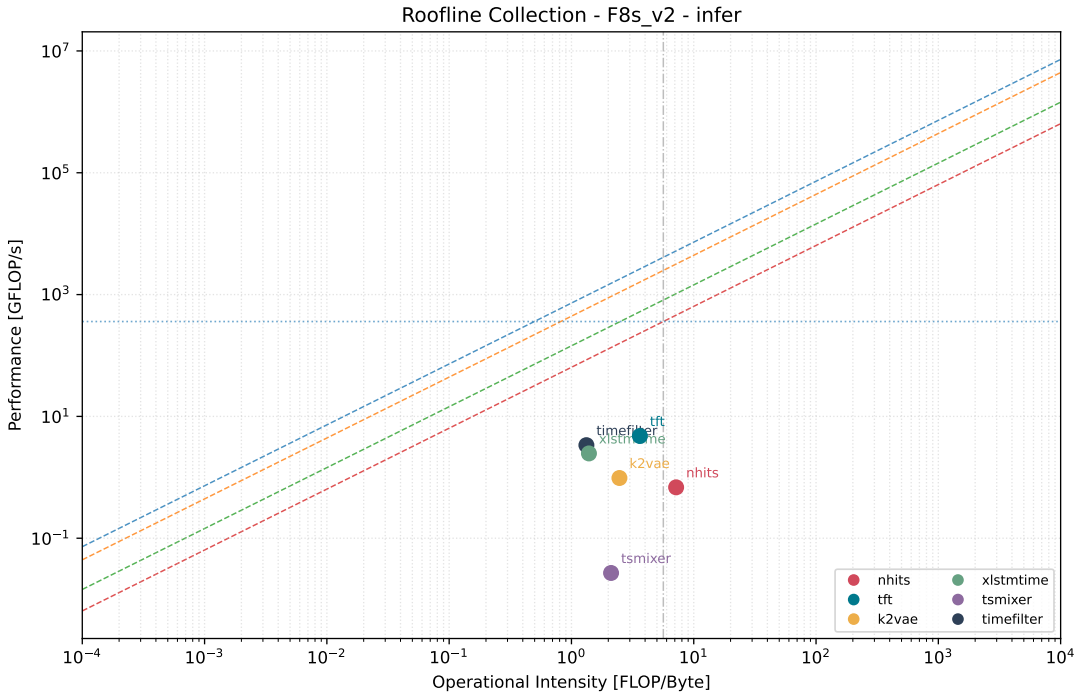
Per-Machine Rooflines

D.1 Empirical Rooflines

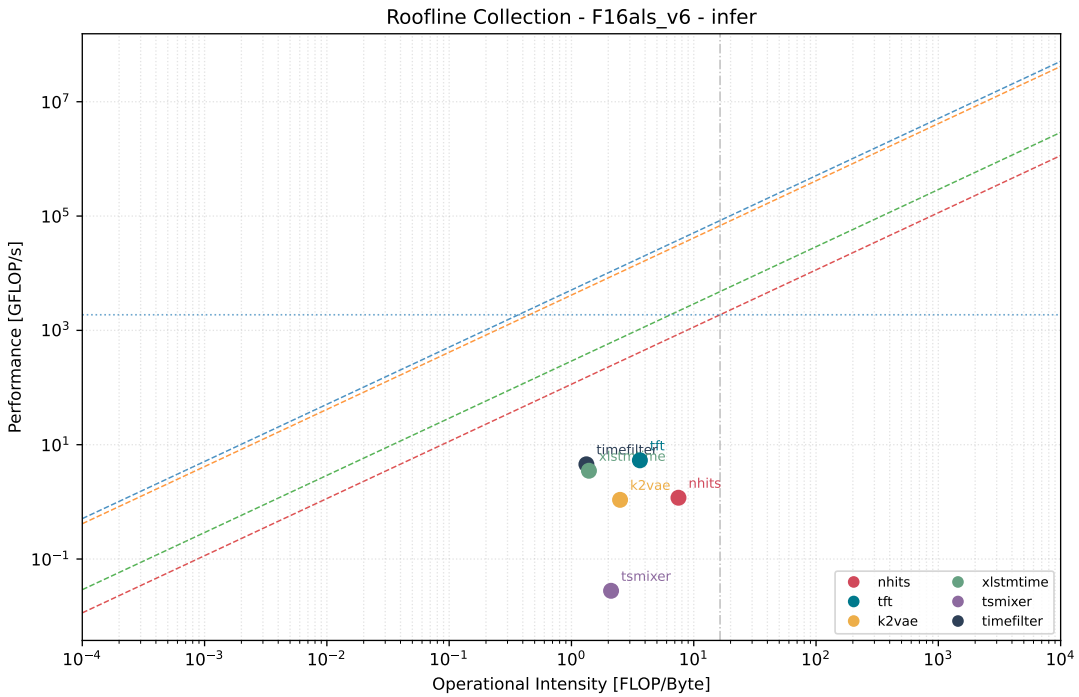
D.1.1 F8als_v6



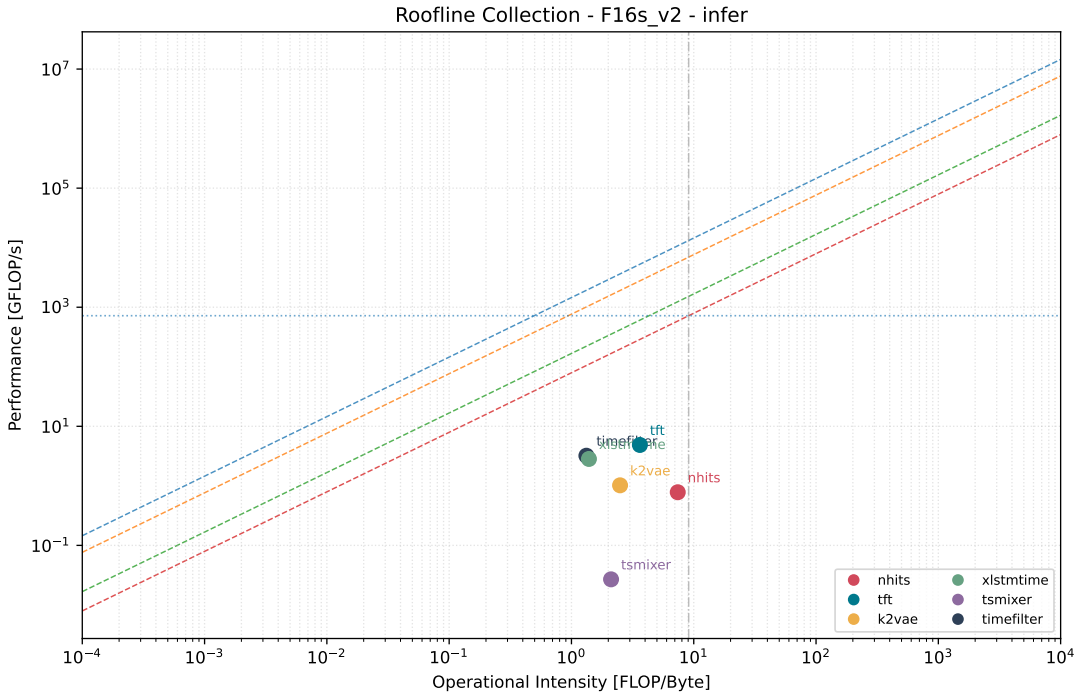
D.1.2 F8s_v2



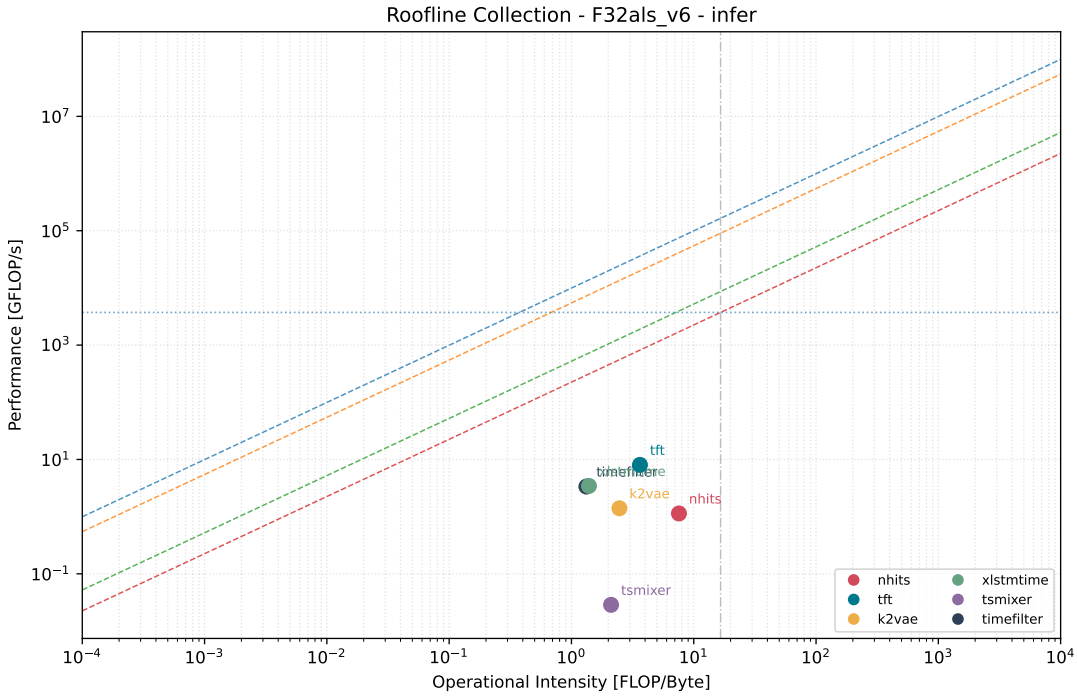
D.1.3 F16als_v6



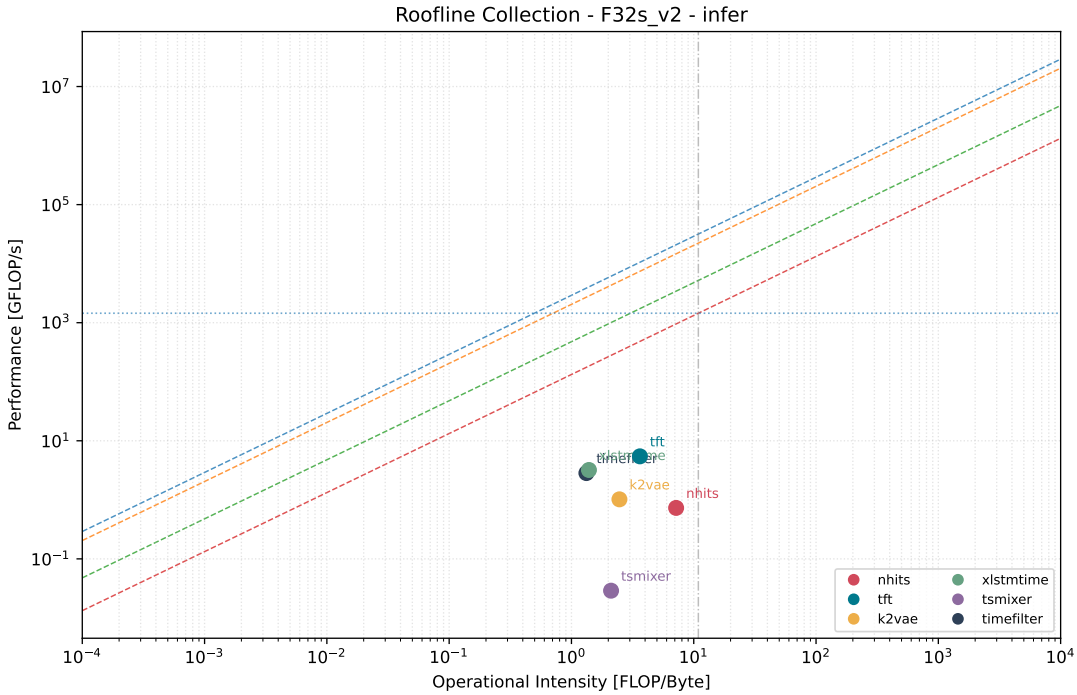
D.1.4 F16s_v2



D.1.5 F32als_v6

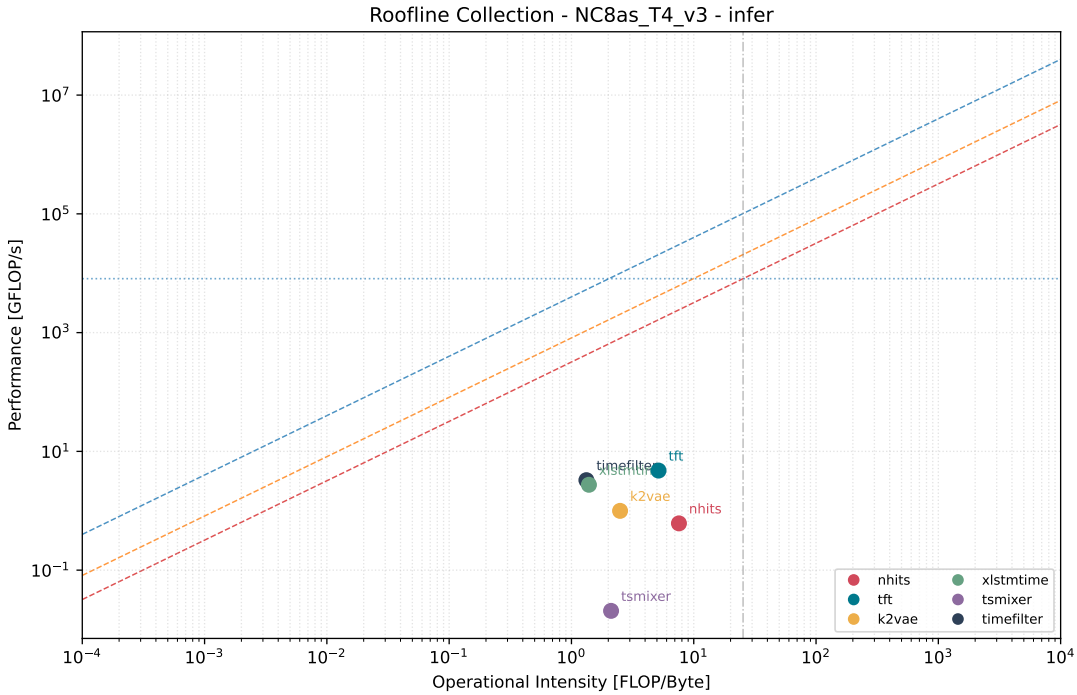


D.1.6 F32s_v2

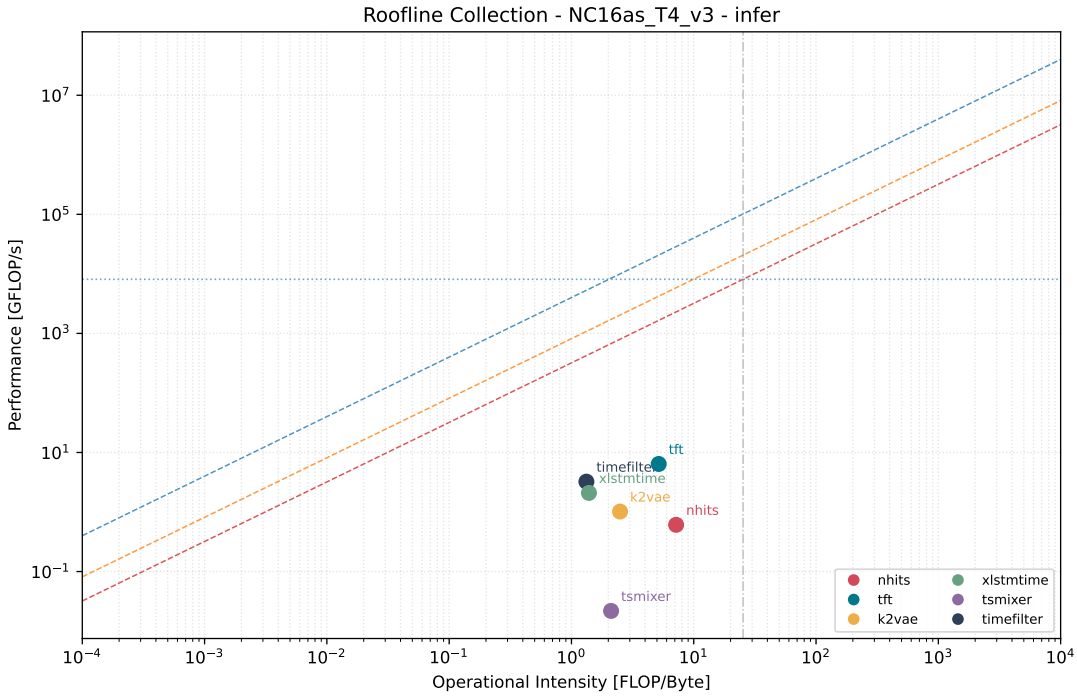


D.2 Theoretical GPU Rooflines

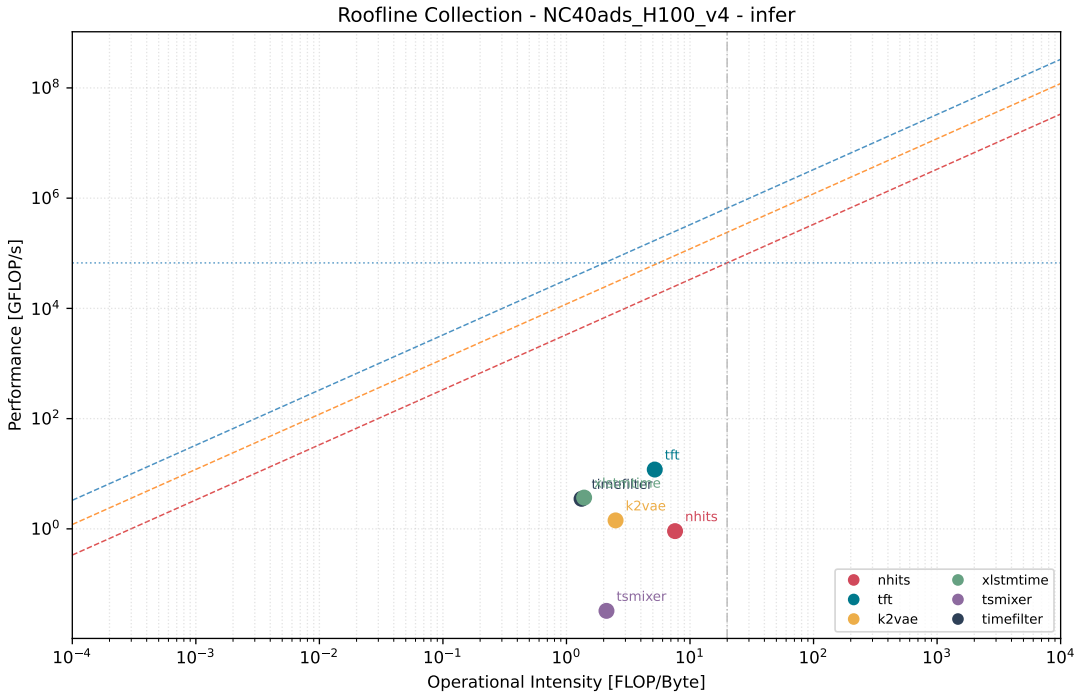
D.2.1 NC8as_T4_v3



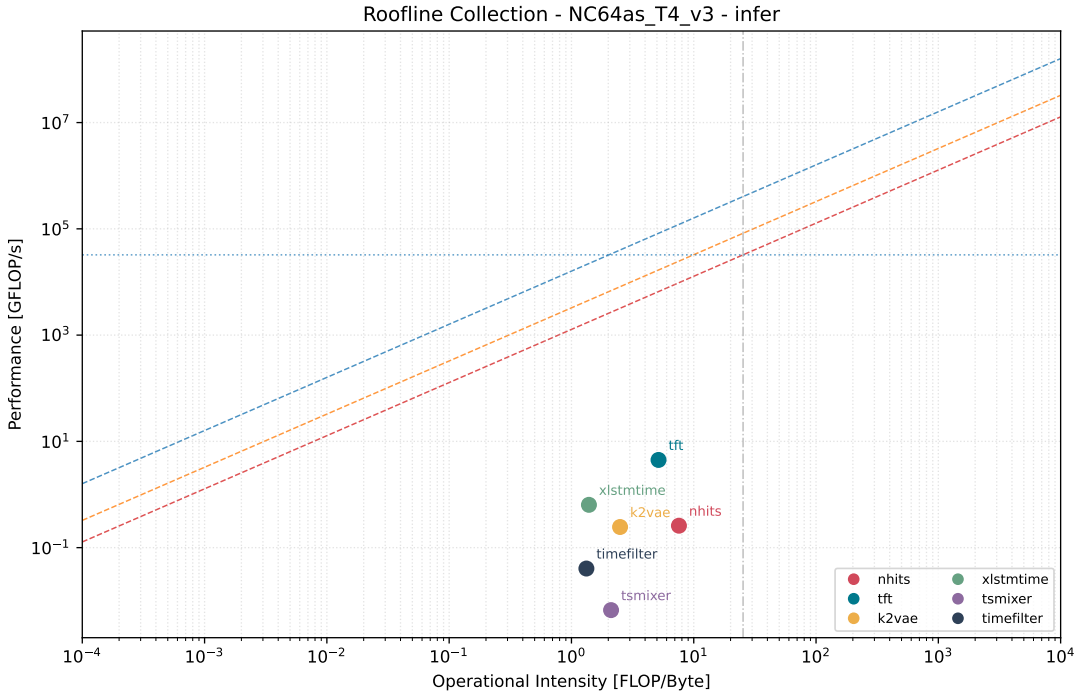
D.2.2 NC16as_T4_v3



D.2.3 NC40ads_H100_v4



D.2.4 NC64as_T4_v3



D.2.5 NC80adis_H100_v4