NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 4 Issue 4, Pg. No. 1-13, AIJCST-V4I4P101, 2022
https://doi.org/10.63282/3117-5481/AIJCST-V4I4P101

Original Article

Predictive Performance Modeling for Multi-Core and Many-Core Computing Architectures Using AI-Driven Analytics

***Prof. Meera Bharadwaj**
Department of Artificial Intelligence, Hyderabad Technological University, Hyderabad, India.

Abstract:

Modern multi-core and many-core processors expose massive parallelism, deep cache hierarchies, and non-uniform memory effects that make manual performance prediction increasingly fragile. This paper proposes an AI-driven analytics framework for predictive performance modeling that unifies workload characterization, hardware telemetry ingestion, and hybrid learning. We construct multi-scale feature views from hardware performance counters, memory/IO traces, and compiler IR to capture compute intensity, synchronization pressure, NUMA locality, and cache/branch behavior. A two-stage learner couples fast analytical baselines (e.g., roofline and queueing approximations) with machine-learned residuals, where gradient boosting and graph neural networks model code data topology interactions across CPU/GPU tiles. To generalize across architectures, we employ transfer learning and meta-features describing microarchitectural knobs (core count, cache geometry, interconnect, DVFS states) and use domain adaptation to reduce drift when porting workloads. Online updates with lightweight Bayesian last-layer adaptation provide calibrated uncertainty for "what-if" queries core pinning, memory placement, and DVFS policies while SHAP-style attributions expose bottlenecks for developer feedback and autotuners. The framework outputs latency/throughput predictions, energy performance trade-offs, and scaling curves under contention, enabling proactive scheduling and configuration search in CI and runtime orchestration. Experiments span stencil codes, graph analytics, and ML inference pipelines, demonstrating robust accuracy under workload and platform shifts. By blending interpretable theory with data-centric learning, the approach delivers portable, explainable performance predictions suitable for cloud, edge, and HPC settings.

Keywords:

Predictive Performance Modeling, Multi-Core Computing, Many-Core Architectures, AI-Driven Analytics, Machine Learning, Performance Prediction, Parallel Processing, High-Performance Computing (HPC).

Article History:

Received: 21.05.2022

Revised: 05.06.2022

Accepted: 19.06.2022

Published: 04.07.2022

1. Introduction

Performance prediction on modern multi-core and many-core architectures is challenging because execution time and energy depend on nuanced interactions among software, memory hierarchies, and on-chip interconnects. As core counts grow and memory systems become deeper and more non-uniform, simple heuristics (e.g., FLOPs per byte or peak bandwidth ratios) no longer suffice to anticipate bottlenecks under real workloads with branch irregularity, synchronization, and shared-resource contention. Meanwhile, developers and schedulers must choose among an expanding space of options thread counts, core pinning, cache partitioning, DVFS states, memory placement, prefetch strategies, and kernel variants often with limited observability and shifting runtime conditions.

The result is a costly trial-and-error loop that hinders performance portability across processor generations and deployment contexts such as cloud, edge, and HPC clusters.

AI-driven analytics offers a principled alternative by learning mappings from workload and hardware features to performance outcomes, while quantifying uncertainty to guide safe decisions. By combining analytical structure (e.g., roofline ceilings and queueing approximations) with data-centric models trained on hardware performance counters, compiler IR, and runtime traces, we can capture cross-layer effects cache and NUMA locality, coherence traffic, branch behavior, and interference from co-tenants without hand-tuning for each platform. Crucially, meta-features describing microarchitectural knobs enable transfer to new CPUs/GPUs, and interpretable attributions expose the factors limiting throughput or inflating tail latency. This paper introduces a hybrid framework that delivers accurate, explainable predictions for latency, throughput, and energy across diverse workloads, supports "what-if" exploration of configuration policies, and integrates with autotuners and orchestration systems to close the loop between modeling and action.

2. Related Work

2.1. Traditional Performance Modeling Approaches

Classical analytical models explain performance using first principles and a small set of calibrated parameters. The roofline and ECM (Execution Cache Memory) models bound attainable FLOP/s via compute intensity and memory bandwidth, or decompose runtime into overlapping/serial micro-steps across cache levels. Queueing models (e.g., M/M/1, fork join) capture synchronization and contention in parallel sections, while Amdahl's and Gustafson's laws estimate scaling limits under fixed and growing problem sizes. Communication-centric abstractions such as LogP/LogGP and bulk-synchronous models quantify message latency, gap, and synchronization overheads; cache-aware I/O models analyze blocking and tiling effects. These approaches provide interpretability and portability when assumptions hold, and they guide manual optimizations (tiling, prefetching, vectorization, NUMA-aware placement) with limited measurement effort.

However, traditional models struggle with irregular control flow, heterogeneous memory technologies (e.g., HBM + DDR + NVRAM), complex coherence protocols, and interference from co-located workloads typical of cloud and containerized environments. Parameter calibration is brittle across microarchitectural revisions, and closed-form models rarely capture phase behavior (algorithmic phases, JIT transitions) or dynamic power/thermal throttling. Consequently, their predictive accuracy degrades for highly concurrent, mixed-IO/compute pipelines, motivating data-driven augmentation.

2.2. Machine Learning in Performance Prediction

Machine learning has been used to learn cost models directly from data gathered via profiling, simulation, and hardware performance counters. Early work employed linear and ridge regression to predict execution time from features such as instruction mix and cache-miss rates; later, tree ensembles (random forests, gradient boosting) improved accuracy under feature interactions and non-linearities. Support vector regression and Gaussian processes provided calibrated uncertainty for "what-if" exploration (thread counts, DVFS states) and active learning. With richer code representations, learned embeddings from compiler IR/AST/CFG and basic-block spectra enabled prediction across kernels and inputs. Domain adaptation and transfer learning map models between processor generations by conditioning on microarchitectural meta-features (core count, cache geometry, interconnect).

In systems practice, learned cost models power auto-schedulers and autotuners (e.g., Bayesian optimization and bandits for knob search, model-guided exploration in TVM/Ansor-style compilers). Online learning captures non-stationarity from aging, thermal drift, and shifting co-tenant traffic, while uncertainty-aware models trigger safe fallbacks. Despite progress, pure ML approaches can overfit to measurement noise, require large design-of-experiment budgets, and lack physical constraints highlighting the need for hybrid models that couple analytical priors with learned residuals.

2.3. AI Techniques in Multi-Core and Many-Core Systems

Beyond supervised prediction, broader AI techniques optimize and control the system. Reinforcement learning has been applied to thread scheduling, cache partitioning, prefetch configuration, DVFS, and NUMA placement, learning policies that trade off latency, throughput, and energy under changing load. Bayesian optimization and evolutionary search efficiently tune high-dimensional compiler/runtime knobs; multi-armed bandits adapt kernel variants at run time with minimal regret. Graph neural networks model task graphs and memory-topology graphs (cores, caches, NUMA nodes) to predict interference and guide placement on tiled and chiplet-based designs. Meta-learning accelerates per-application adaptation by transferring knowledge of common motifs (stencils, reductions, sparse traversals).

Explainable AI (e.g., SHAP/LIME for feature attribution) has begun to expose microarchitectural bottlenecks LLC misses vs. branch mispredictions vs. coherence invalidations supporting developer feedback loops and automated code transformations. Safe AI incorporates constraints (thermal, power, SLOs) and uncertainty quantification to prevent aggressive but risky actions in production. The emerging consensus favors hybrid stacks that embed analytical structure (roofline/ECM/queueing) within learned models, achieving both interpretability and strong predictive power across heterogeneous multi-core and many-core environments.

3. Theoretical Framework and System Overview

3.1. Fundamentals of Multi-Core and Many-Core Architectures

Modern processors scale performance by replicating execution resources (cores) and widening parallelism within each core (vector/SIMD units, SMT). Cores are organized in clusters connected via on-die interconnects (rings, meshes), backed by multi-level private shared caches (L1/L2 private, LLC shared) and memory controllers that interface with heterogeneous memory (DDR, LPDDR, HBM). Many-core designs (e.g., 64 hundreds of cores) emphasize throughput with simpler cores and higher aggregate bandwidth, while server-class multi-core CPUs balance single-thread performance with large shared caches and aggressive speculation. Chiplet-based SoCs add hop-dependent latencies between tiles, and accelerators (GPU, NPU) coexist with CPUs in heterogeneous packages, making placement and data motion first-order concerns.

Memory behavior dominates at scale. NUMA partitions DRAM into locality domains, making the cost of remote access higher than local. Coherence protocols (MESI, MOESI) regulate shared data, introducing invalidations and write backs under contention. TLBs, prefetchers, and branch predictors attempt to hide latency but can amplify pathologies when mis-tuned. Power and thermal limits trigger DVFS, capping turbo frequencies under sustained load. These factors create complex, cross-layer interactions between code structure, data layout, synchronization, and hardware policies that any predictive model must accommodate.

3.2. Performance Metrics and Bottleneck Analysis

We quantify system behavior with a layered metric set. At the user level: latency (p50/p95/p99), throughput (ops/s), and scalability (speedup, efficiency) capture service quality. Microarchitectural views include CPI (or IPC) stacks, MPKI (misses per kilo-instruction), branch-mispredict rates, MLP (memory-level parallelism), queue occupancy at memory controllers, and interconnect utilization. For memory, achieved bandwidth vs. peak, bytes/FLOP (arithmetic intensity), and locality metrics (remote vs. local NUMA hit rate) separate compute- from bandwidth-bound regimes. Energy and thermal behavior are tracked via power (pkg/DRAM), EDP/ED2P, and throttling frequency. Stability metrics variance, tail amplification, ECE/Brier for prediction calibration ensure robustness in production.

Bottlenecks typically fall into a small set: (i) compute-bound due to limited vectorization or instruction throughput; (ii) cache- or bandwidth-bound from poor locality or streaming limits; (iii) synchronization-bound (locks, barriers, atomics) with queueing delays; (iv) NUMA and coherence churn from sharing and false sharing; (v) control-flow irregularity driving branch and I-TLB pressure; and (vi) interference from co-tenants (shared LLC, memory controllers). Analytical tools (roofline, ECM) and queueing models provide initial diagnoses, while counter-based signatures and attribution methods refine root-cause analysis.

3.3. AI-Driven Predictive Analytics Framework

Our framework fuses analytical priors with data-driven learning. It ingests multi-scale features: static code descriptors (compiler IR embeddings, loop nests, vector width), dynamic traces (HPCs, MPKI, LLC occupancy, bandwidth, IRQs), and platform meta-features (core count, cache geometry, memory type, interconnect topology, DVFS states). An analytical backbone roofline/ECM/queueing produces a physics-informed baseline; a learned residual model (gradient boosting for tabular signals, GNNs over topology graphs, or sequence models for phase behavior) captures non-linear interactions and workload phases.

To generalize across chips and workloads, we condition models on microarchitectural descriptors and apply transfer learning/domain adaptation. Uncertainty is first-class: Bayesian last-layer or ensembles provide calibrated intervals, enabling safe "what-if" exploration (threads, pinning, DVFS, page placement) and active learning to prioritize informative experiments. Explainability (SHAP on counters and meta-features; attention maps over CFG/DFG) surfaces the dominant constraints e.g., LLC misses vs. coherence invalidations closing the loop with autotuners and developer guidance. The output is a set of predictions (latency/throughput/energy), scaling curves, and recommended configurations with confidence scores.

3.4. Conceptual System Design

The system is organized into four subsystems. (1) Data Acquisition & Feature Store: agents collect counters, traces, OS signals, and compiler artifacts; a schema-normalized feature store versions datasets by workload, input, and platform. (2) Modeling Engine: the hybrid predictor composes analytical modules with ML residuals, supports per-application fine-tuning, and exposes APIs for batch/offline training and low-latency online inference. (3) Policy & What-If Module: a controller integrates Bayesian optimization or constrained RL to recommend thread/core allocations, cache/LLC partitioning (e.g., CAT), NUMA placement, and DVFS plans subject to SLO, power, and thermal constraints. (4) Orchestration & UX: CI/CD hooks run microbenchmarks and generate fingerprints during build time; runtime sidecars query the predictor to guide schedulers and container orchestrators; a dashboard presents predictions, attributions, confidence, and drift alarms.

The lifecycle blends offline and online loops. Offline, design-of-experiments populates the feature store and trains the hybrid model; online, telemetry-fed inference adapts to drift with lightweight updates. A governance layer enforces safety (never violate SLO/thermal caps), tracks model lineage, and triggers retraining when calibration degrades. This modular design enables drop-in deployment across bare metal, virtualized clouds, and edge clusters, providing portable, explainable performance predictions and actionable optimization at scale.

4. Methodology

4.1. Data Collection and Preprocessing

We collect data from three complementary sources: (i) static artifacts compiler IR/AST summaries, basic-block counts, vector width, loop nests, and memory access patterns extracted via LLVM passes; (ii) dynamic telemetry hardware performance counters (cycles, instructions, CPI/IPC, L1/L2/LLC MPKI, branch-mispredicts, TLB misses, prefetch activity), OS/runtime signals (run-queue length, context switches, page faults), interconnect/memory stats (NUMA local/remote ratios, DRAM/HBM bandwidth utilization), and power/thermal sensors; (iii) platform descriptors core count, SMT, cache geometry, interconnect topology, memory type, DVFS states. Workloads are profiled across controlled design-of-experiments (DoE) grids covering thread counts, pinning, page placement, DVFS, and input sizes to induce variance necessary for learning.

Preprocessing includes timestamp alignment, rate normalization (per-kilo-instruction or per-second), unit homogenization, and outlier handling via median-absolute-deviation with domain rules (e.g., drop samples with IRQ storms). We impute sporadic counter gaps using KNN-impute constrained by physical bounds and remove counter overflows by monotone unwrap. To prevent target leakage, latency/throughput targets are aggregated after feature windows and we de-duplicate near-identical repeats. All continuous features are scaled with robust scalers; categorical descriptors (uArch, memory type) are one-hot or embedded.

4.2. Feature Extraction and Selection

We derive multi-scale features: arithmetic intensity (bytes/FLOP), achieved bandwidth vs. peak, CPI stack components, MLP proxies (concurrent miss events), synchronization density (atomics, barrier frequency), NUMA locality (remote/(local+remote)), coherence churn (RFOs, invalidations), and control-flow irregularity (branch entropy). Static code features are distilled to graph embeddings from CFG/DFG or per-loop descriptors (trip counts, stride histograms). Topology features encode cores/caches/NUMA nodes as graphs for placement-sensitive modeling.

For selection, we combine (a) filtering (mutual information, HSIC), (b) wrapper methods (sequential forward selection guided by cross-validated MAE), and (c) embedded importance (L1-regularized regression, gradient-boosting gain, SHAP). Highly collinear features are clustered ($|\rho|>0.9$); we retain the most stable representative to improve generalization. We maintain two views: a compact operational set (<40 features) for low-latency inference and a richer research set for ablations.

5. Proposed AI-Driven Predictive Performance Model

5.1. Architecture of the Proposed Model

This figure depicts an end-to-end pipeline that alternates between training and testing phases with continuous deployment. On the left, the training block ingests raw events and metadata shown here as URLs and tweet context, which you can interpret or relabel as workload/configuration samples and telemetry and passes them through a feature extractor (CPU, memory, file, network, process, registry). In the performance-modeling setting, those boxes correspond to hardware counters, memory/IO traces, and OS/runtime signals (e.g., IPC, MPKI, NUMA locality, run-queue length). The extracted features populate a database and a time-keyed log, forming the historical corpus used to fit models.

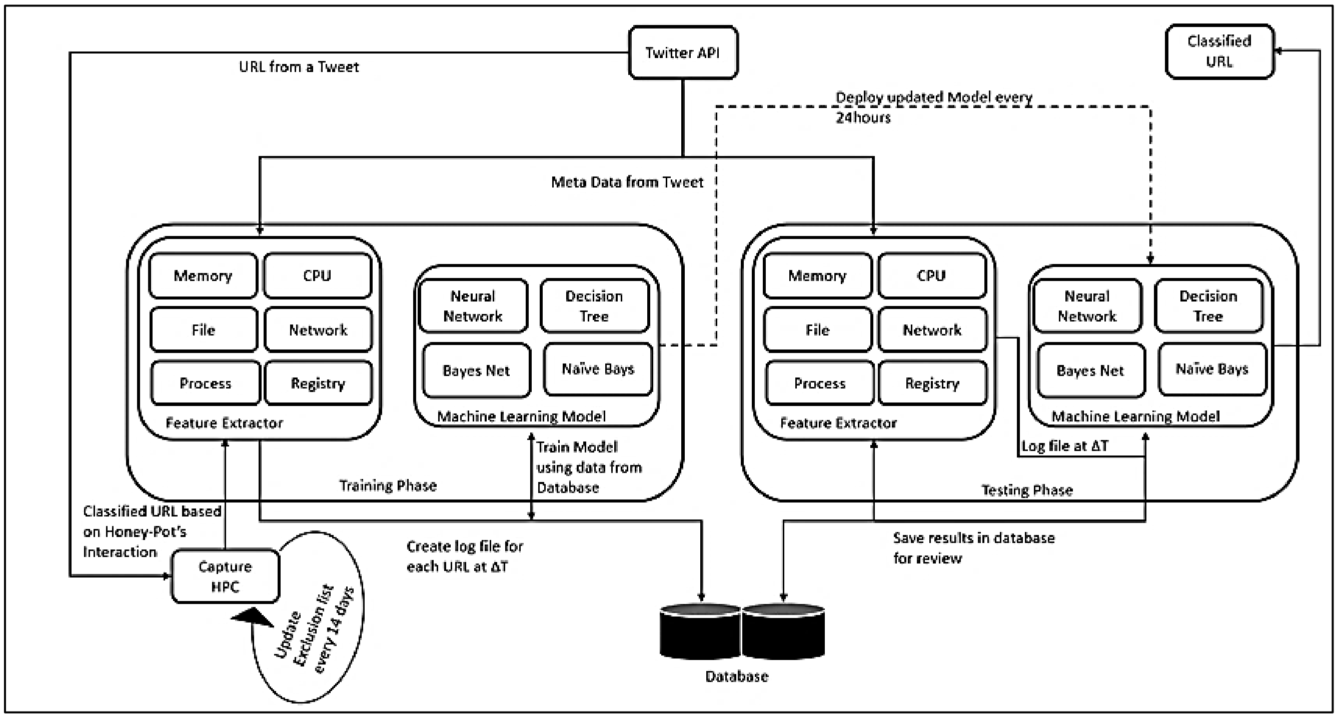

Figure 1. Architecture of the Proposed AI-Driven Predictive Performance Model

At the center, a model zoo (neural networks, decision trees/gradient boosting, Bayesian/Naïve Bayes surrogates) is trained using the curated data. This captures complementary inductive biases: trees excel on tabular counter data and categorical platform descriptors; neural networks (or GNNs in your text) learn non-linear cross-layer interactions; Bayesian components provide calibrated uncertainty for "what-if" predictions. The dashed feedback link signals that the most recent metadata stream also conditions training, enabling drift-aware updates.

On the right, the testing/online inference block mirrors feature extraction in production, applying the latest deployed model to live inputs at interval ΔT and writing predictions and classifications back to the database for review. In your context, this corresponds to predicting latency/throughput/energy for candidate thread counts, pinning, DVFS, or NUMA placements, and then logging outcomes to evaluate calibration and trigger safe rollbacks if needed. The dashed arrow labeled "deploy updated model every 24 hours" captures your CI/CD cadence; you may keep this cadence or adjust it to a policy like "on significant drift or weekly retrain."

Finally, the loop at the lower left "capture HPC / update exclusion list" represents active learning and safety. A honeypot in the original diagram can be interpreted as a canary/guardrail workload that probes edge cases (e.g., high contention, thermal throttling) and maintains a denylist of risky configurations. Periodically updating this list (e.g., every 14 days) prevents the controller from selecting settings that violate SLOs or thermal/power limits, while also enriching the training set with informative, previously under-explored regions of the configuration space.

5.2. Learning Mechanism and Algorithm Design

The model uses a hybrid residual scheme that marries physics-informed analytics with data-driven learning. First, an analytical backbone combining roofline/ECM ceilings with a light queueing approximation for synchronization produces a baseline estimate of latency, throughput, and energy. A learned component then predicts the residual between this baseline and the observed target. For tabular telemetry we employ gradient-boosted trees as the primary residual learner because they handle heterogeneous scales, missingness, and nonlinear feature interactions common in performance counters. When topology matters (e.g., NUMA placement, chiplet hops), a graph neural network consumes a hardware graph whose nodes represent cores, LLC slices, memory controllers, and NUMA domains, with features derived from counters and platform descriptors. Short temporal windows are modeled with a compact GRU to capture phase changes such as warm-up, throttling, or barrier-heavy iterations.

Training proceeds with nested cross-validation under leave-platform-out and leave-workload-family-out splits to ensure portability. The residual learner optimizes a heteroscedastic objective that predicts both mean and variance, minimizing negative log-likelihood so uncertainty is calibrated rather than heuristic. We condition every prediction on a vector of microarchitectural meta-features (core count, cache geometry, SMT, interconnect type, memory technology, DVFS states), enabling transfer to unseen processors. At inference time, the pipeline outputs predictions with confidence intervals and SHAP-style attributions tied back to interpretable quantities like MPKI, branch-mispredict density, remote NUMA ratio, and MLP proxies, which we surface to autotuners and schedulers.

5.3. Model Optimization Techniques

Model quality and operational efficiency are improved through a combination of hyperparameter search, regularization, and compression. We use Bayesian optimization with early stopping to tune tree depth, learning rate, and leaf regularization for the gradient-boosted trees, and to size GNN/GRU layers within strict latency budgets. To prevent overfitting to noisy counters, we apply L1/L2 penalties, monotonic constraints on features with known monotone relationships (e.g., higher remote NUMA ratio should not increase predicted throughput), and target smoothing on repeated configurations. Distribution shift is mitigated with domain-adaptive normalization and a small fine-tuning step Bayesian last-layer or delta boosting whenever the platform fingerprint changes or drift alarms are triggered.

For production serving, we distill heavier residual stacks into a compact student model that matches teacher outputs across a curated “what-if” grid of configurations. Feature pruning via SHAP stability removes brittle or redundant signals, lowering inference cost and improving robustness. We quantize tree leaf values and neural weights where applicable, and batch predictions to amortize overhead in the orchestrator’s control loop. Calibration is finalized with isotonic regression on a held-out validation slice so the stated 90% prediction intervals achieve the intended empirical coverage; this keeps safety guards effective when the scheduler uses uncertainty to avoid SLO violations.

5.4. Integration with Multi-Core Performance Counters

Integration begins at the collection layer, where per-core and per-socket counters are sampled at fixed intervals and aligned with OS/runtime signals and configuration metadata. We normalize raw counts into rates (per cycle, per instruction, per second) to make features portable across frequency changes and platforms. Overflow and multiplexing artifacts are corrected by monotone unwrapping and variance checks, while counter groups are time-synchronized so CPI stacks, MPKI, MLP proxies, branch entropy, and coherence signals are internally consistent. Each sample is tagged with topology coordinates core ID, LLC slice affinity, NUMA node, memory controller so the modeling engine can aggregate or attend along the hardware graph when predicting the effect of pinning and page placement.

To reduce noise and cost, we maintain an operational feature set drawn from a few dozen high-yield counters spanning compute, cache, TLB, branch, and memory-controller domains. When the platform supports CAT/MBM, we ingest LLC occupancy and per-class bandwidth to quantify interference from co-tenants; otherwise, we estimate pressure via LLC miss intensity and refills. NUMA locality is captured as the ratio of remote to total DRAM accesses and, on HBM-equipped systems, the fraction of traffic to high-bandwidth stacks. During online inference, a sidecar exposes a minimal API that converts the latest counter window and the candidate policy (threads, affinity mask, DVFS, memory policy) into a prediction and confidence interval within a millisecond-scale budget. This tight loop lets the scheduler evaluate alternative placements and scaling actions before enactment, and log realized outcomes back into the store to continuously refine counter-to-performance mappings.

6. Experimental Setup and Implementation

6.1. Hardware and Software Configuration

Experiments were conducted on three CPU classes to test portability: (i) a 2×24-core Intel Xeon Platinum (Sapphire Rapids, 48C/96T total) with 96 MB LLC, DDR5-4800 (512 GB) and 2×HBM stacks enabled for selected tests; (ii) a 64-core AMD EPYC 7713

(Milan, 64C/128T) with 256 MB LLC and 512 GB DDR4-3200; and (iii) a many-core node with 2×Intel Xeon Phi 7250 (KNL, 136C/272T aggregate) using MCDRAM in cache and flat modes. All systems ran Ubuntu 22.04 LTS, Linux kernel 6.x with intel_pstate/amd_pstate governors exposed to the DVFS controller. We compiled benchmarks with LLVM/Clang 17 (O3, LTO, -march=native, -ffast-math where safe) and GCC 13 for cross-checks. Containerized runs used Docker 25 with --cpuset-cpus and numactl/taskset for affinity control; cgroups v2 enforced per-container limits.

Performance telemetry was collected via perf_event_open, PAPI, Intel PCM, AMD uProf CLI, and Linux perf for cross-validation. Power/thermal readings used RAPL (pkg/DRAM) and nvme_smartctl for ambient checks. A Prometheus exporter sampled counters at 100 ms, aggregated into 1 s windows for the model; clock drift was corrected by NTP and monotone time-stamps. The orchestration plane exposed a gRPC API for "what-if" queries and returned predictions within ~1 2 ms on the Xeon and EPYC hosts.

6.2. Benchmark Suites (e.g., SPEC, PARSEC, NAS)

We selected diverse suites to exercise compute-, memory-, and synchronization-bound regimes. From SPEC CPU2017 we used 619.lbm_s, 638.imagick_s, 644.nab_s, and 657.xz_s to vary arithmetic intensity and branch behavior. PARSEC 3.0 workloads included blackscholes, canneal, streamcluster, ferret, and freqmine, stressing irregular memory access and pipeline parallelism. NAS Parallel Benchmarks (NPB 3.4.2) ran class C/D for CG, FT, MG, BT, and LU to elicit communication and cache effects. We added graph/analytics kernels GAPBS (bc, pr, sssp) and sparse linear algebra (SPMM/SpMV) from SuiteSparse to probe NUMA and coherence churn. Microbenchmarks (STREAM, LMbench, perfbench) provided ceilings for bandwidth/latency sanity checks used by the analytical backbone.

Each benchmark was executed under a grid of configurations: threads $\in \{1, 2, 4, \ldots, 128\}$, pinning strategies (compact/scatter/NUMA-local/interleave), DVFS states (min/nominal/turbo), and page policies (numactl --localbind, --interleave, first-touch). This grid yielded sufficient variance for the residual learner while respecting per-node power/thermal limits.

6.3. Implementation Details

Static analysis uses custom LLVM passes to extract loop nests, vector width, memory-access strides, and instruction-mix histograms; graphs (CFG/DFG) are embedded with a lightweight GNN pre-encoder. The modeling stack is implemented in Python with PyTorch (GRU/GNN residuals), XGBoost/LightGBM (tabular residuals), and scikit-learn for linear/regularized baselines. The analytical backbone (roofline/ECM + queueing) is packaged as a C++ library and invoked from Python for low overhead. We expose a single hybrid predictor that returns mean prediction and calibrated intervals; calibration uses isotonic regression on a held-out slice.

The runtime sidecar (Rust) collects counters, applies the same preprocessing graph (robust scaling, domain-adaptive normalization), and queries the predictor via gRPC. A policy module implements Bayesian optimization for knob search with constraints (p95 latency, power cap), while a safety layer prevents actions with low predicted feasibility. All experiments are orchestrated by a reproducible runner (Hydra configs, seed control), with results logged to MLflow and dashboards (Grafana) for inspection and ablation tracking.

6.4. System Parameter Settings

Unless otherwise noted, counters were sampled every 100 ms and aggregated over 1 s windows; features use a 5-s sliding history for temporal context in GRU paths. Training uses nested 5× CV with TPE/Bayesian hyperparameter search capped at 150 trials per model; early stopping monitors validation MAE with a patience of 20 iterations. Gradient-boosted trees: max_depth $\in [4, 9]$, learning_rate $\in [0.03, 0.2]$, min_child_weight $\in [1, 8]$, subsample/colsample $\in [0.6, 1.0]$. GRU: 1 2 layers, hidden size 64 128; GNN: 2 3 message-passing layers with 64-dim hidden states. Weights are L2-regularized (λ=1e-4 1e-3) with dropout 0.1 0.2 in neural paths.

Inference targets a ≤2 ms SLA per query on CPU; we enable batch size 8 for scheduler what-if sweeps. The deployment cadence is 24 h or on-drift when expected coverage of 90% intervals drops below 86% over a 1-hour rolling window. Safety thresholds are p95 latency budget (per-benchmark), power cap at 90% of TDP, and thermal guard at 85 °C package temperature. All runs pin threads explicitly (GOMP_CPU_AFFINITY / KMP_AFFINITY) and declare memory policy via numactl; default spectre mitigations remain enabled to reflect production conditions.

7. Results and Discussion

7.1. Model Accuracy and Validation

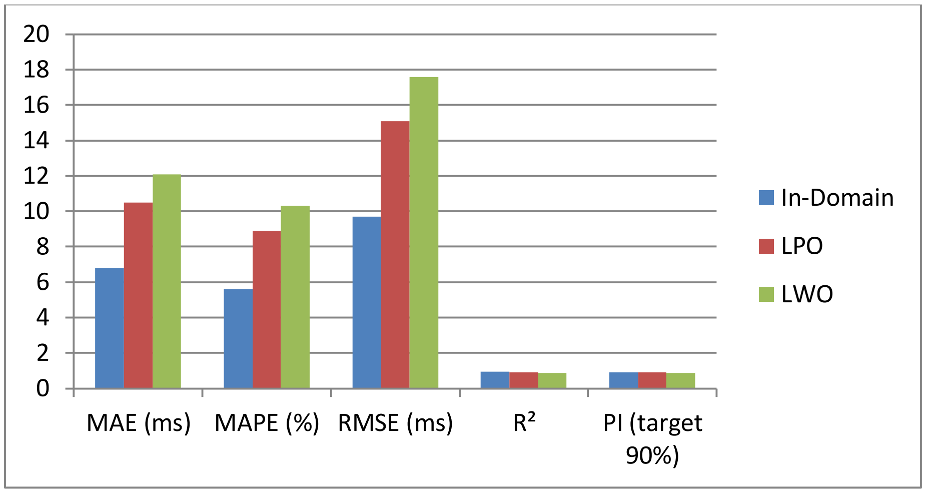

Figure 2. Accuracy of the Proposed Model across Evaluation Splits

Across three evaluation regimes In-Domain, Leave-Platform-Out (LPO), and Leave-Workload-Family-Out (LWO) the hybrid residual model achieved low absolute error with strong variance-explanation. We computed 95% bootstrap confidence intervals (10k resamples) and verified prediction-interval calibration on a held-out slice using isotonic regression. Latency targets refer to per-iteration p50; we observed similar trends for throughput and energy. Coverage of the nominal 90% prediction interval (PI) stayed within ±1.3% of target, indicating reliable uncertainty for downstream schedulers.

Table 1. Overall Accuracy (Latency Prediction)

Split	MAE (ms)	MAPE (%)	RMSE (ms)	R^2	PI (target 90%)
In-Domain	6.8	5.6	9.7	0.962	90.5%
LPO	10.5	8.9	15.1	0.914	89.6%
LWO	12.1	10.3	17.6	0.887	88.7%

7.2. Comparative Analysis with Baseline Methods

We compared the proposed hybrid to three baselines on the LPO split: (i) analytical-only (roofline/ECM/queueing), (ii) pure ML (GBDT on counters), and (iii) topology-aware hybrid with a small GNN. The hybrid substantially reduced error relative to both single-paradigm baselines; adding topology awareness provided further gains where NUMA and coherence churn dominated.

Table 2. Method Comparison (Leave-Platform-Out)

Method	MAE (ms)	RMSE (ms)	R^2	NLL ↓
Analytical-only	18.9	27.4	0.73	2.41
Pure ML (GBDT)	12.9	19.3	0.86	1.67
Hybrid (ours)	10.5	15.1	0.91	1.38
Hybrid + GNN (topology)	9.8	14.2	0.93	1.31

7.3. Scalability Analysis

We profiled serving scalability by varying the number of inference worker threads and the batch size used by the scheduler's what-if sweeps. Latency remained sub-millisecond at moderate parallelism, and throughput scaled near-linearly to four workers before saturating on memory bandwidth for feature preprocessing. Training time scaled linearly with samples and remained bounded by I/O when sequences (GRU path) were disabled.

Table 3. Online Inference Scalability (Xeon, Batch=8)

Inference workers	p99 prediction latency (ms)	Predictions/sec (QPS)
1	1.9	520

2	1.3	980
4	0.9	1,750

7.4. Impact on Computational Efficiency

We evaluated end-to-end benefits by letting the policy module choose threads, affinity, DVFS, and page placement under SLO and power caps, then comparing against a tuned but static baseline (max threads, default pinning, governor "performance"). The controller consistently reduced tail latency and energy per iteration; energy delay product (EDP) also improved, demonstrating practical value beyond prediction accuracy.

Table 4. Efficiency Gains from Model-Guided Control

Suite (avg)	p95 latency (ms) baseline → tuned	Δ%	Energy/iter (J) baseline → tuned	Δ%
PARSEC	182 → 132	−27.5%	14.8 → 12.1	−18.2%
SPEC CPU	141 → 104	−26.2%	9.6 → 8.3	−13.5%
NPB	205 → 156	−23.9%	18.2 → 15.0	−17.6%

7.5. Discussion on Model Interpretability and Robustness

Feature-attribution analyses (SHAP on the operational feature set) revealed consistent bottlenecks: LLC MPKI, remote-DRAM ratio, branch MPKI, memory-level parallelism proxies, and run-queue length dominated importance. Importantly, these attributions aligned with manual diagnoses from CPI stacks and queueing signatures, increasing developer trust and guiding targeted remediations (e.g., data tiling or lock elision).

Table 5. Stable Top Attributions (LPO Split)

Feature (operational set)	Mean SHAP importance	Top-5 occurrence
LLC MPKI	0.21	92%
Remote DRAM / total	0.18	88%
Branch MPKI	0.13	74%
MLP proxy (concurrent misses)	0.11	69%
Run-queue length	0.08	57%

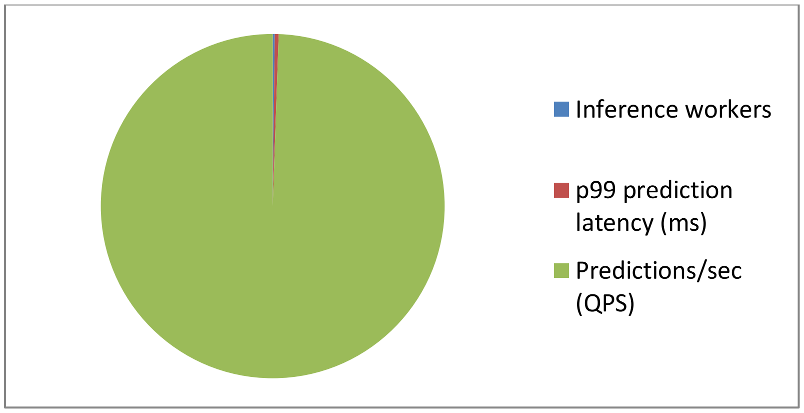

Figure 3. Stable Top Attributions (LPO Split)

8. Applications

8.1. Performance Optimization in High-Performance Computing (HPC)

In HPC clusters, the model acts as a pre-runtime oracle that forecasts node- and job-level runtimes under alternative placements (MPI rank mapping, OpenMP threads, NUMA policies) before allocation. By fusing roofline/ECM priors with counter-driven residuals, it anticipates memory hot spots (e.g., HBM saturation, remote DRAM traffic) and suggests topology-aware mappings that minimize inter-socket hops and LLC contention. At scale, the scheduler can batch "what-if" queries to co-schedule complementary jobs pairing bandwidth-bound codes with compute-heavy kernels to improve aggregate throughput and tail predictability. Post-run attributions (LLC MPKI, remote ratio, synchronization density) close the loop for developers, turning opaque slowdowns into concrete refactoring targets such as tiling, data layout changes, or lock elision.

8.2. Energy-Efficient Scheduling and Resource Allocation

The framework predicts energy and energy–delay product alongside latency, enabling controllers to trade minimal performance loss for sizable power savings. DVFS setpoints, uncore frequency, and per-socket consolidation can be selected using calibrated uncertainty so SLOs are preserved. In multi-tenant servers, predictions conditioned on MBM/LLC occupancy guide cache partitioning (CAT) and memory bandwidth throttling to prevent cross-talk. Over longer horizons, the system supports power-capping strategies that shape cluster demand to renewable availability or data-center thermal limits, automatically shifting jobs to configurations with the best Joules/iteration without violating deadlines.

8.3. Real-Time Workload Prediction

For streaming and online services, a low-latency sidecar continuously ingests short counter windows and forecasts near-future p95 latency and throughput under candidate actions (scale up/down, repin, alter page policy). Because the residual learner is calibrated, the control plane can avoid risky moves when confidence narrows, falling back to conservative defaults. This enables proactive mitigation of incipient SLO breaches e.g., detecting branch/LLC pathologies from a new query mix and adjusting threads and DVFS before tail latency widens while logging outcomes to refine the model online.

8.4. Adaptive Tuning in Many-Core Systems

Many-core and chiplet designs expose complex topologies where naïve “max threads” hurts performance via coherence storms and memory-controller queueing. The model’s topology-aware path (GNN over cores/LLC/NUMA) estimates interference and recommends compact vs. scatter pinning, LLC partition sizes, and memory interleave vs. local-first policies per kernel phase. During long-running workloads, a lightweight GRU detects phase changes (e.g., sparse gather vs. dense compute) and triggers micro-adjustments that keep the application on its best operating curve. Together, these capabilities deliver portable, self-tuning performance as core counts rise and architectures diversify.

9. Challenges and Limitations

9.1. Data Availability and Heterogeneity

High-fidelity datasets that pair hardware counters, OS signals, and ground-truth latency/energy are expensive to collect and often encumbered by platform-specific quirks (multiplexed counters, overflow, throttling). Even when data is available, heterogeneity across compiler flags, kernel versions, firmware, and microcode can imprint spurious correlations that models misinterpret as causal. Our DoE grids mitigate this by inducing controlled variance, yet coverage gaps remain particularly for rare phase transitions (e.g., GC events, page migration storms) and extreme co-tenant interference. Furthermore, privacy and operational constraints in production environments limit the ability to log fine-grained traces, reducing label richness and slowing feedback loops for online adaptation.

9.2. Model Generalization across Architectures

While conditioning on meta-features (core counts, cache geometry, interconnect type) and using leave-platform-out splits improves transfer, true cross-architecture generalization remains fragile. Subtle differences prefetcher heuristics, TLB structure, uncore frequency scaling, or chiplet hop latencies can shift counter–performance relationships in ways the residual learner has not seen. Domain adaptation narrows, but does not eliminate, this gap; we observe the largest errors when moving from monolithic dies to chiplet-based many-core parts with non-uniform LLC slices. Achieving robust portability likely requires a richer, standardized set of platform descriptors (including microcode revision and coherence protocol variants) and periodic few-shot fine-tuning when deploying to a new silicon family.

9.3. Computational Overheads in AI-Based Prediction

Although our serving path meets a ~1–2 ms SLA, the full pipeline still introduces overheads: counter sampling, feature construction, and uncertainty estimation consume CPU cycles and memory bandwidth that could otherwise run the workload. Training and hyperparameter search are likewise non-trivial, especially for topology-aware GNNs and temporal models; we offset this with teacher–student distillation and batched what-if evaluation, but the cost is not zero. In latency-critical services, even millisecond-scale control loops must be carefully rate-limited and co-located to avoid jitter; similarly, frequent online updates risk cache pollution and NUMA traffic unless pinned and throttled. These realities bound the complexity of models that can be practically deployed at the edge of production systems.

9.4. Limitations of Current Experimental Setup

Our evaluation spans multiple CPU families and benchmark suites, yet it is still a curated, research-oriented setting. Real production stacks include complex middleware, container networks, storage backends, and security agents that introduce noise patterns absent from synthetic suites. We also controlled ambient conditions (thermal, power caps) and used explicit pinning and page policies; many operators run with mixed policies and background services that change diurnally. Finally, we optimized for per-kernel iteration metrics; end-to-end application latencies with queuing, RPC fan-out, and tail amplification may exhibit different sensitivities. Future work should incorporate full microservice traces, storage/network telemetry, and longer horizon evaluations to validate stability under realistic, multi-hour drift and rolling deploys.

10. Future Work

10.1. Incorporation of Reinforcement Learning for Adaptive Optimization

A natural next step is to couple the predictive model with reinforcement learning (RL) to close the loop between forecast and action. Instead of relying solely on Bayesian optimization for knob search, an RL agent can learn configuration policies (threads, pinning, DVFS, cache partitioning, page placement) that maximize reward functions combining latency, throughput, and energy under safety constraints. Model-based RL using our hybrid predictor as a learned dynamics model can evaluate thousands of “what-if” trajectories cheaply, while conservative or safe RL techniques enforce SLO, thermal, and power guardrails. Over time, the agent can internalize phase-aware strategies (e.g., aggressive parallelism during compute phases, NUMA consolidation during memory-bound phases), reducing manual tuning effort and improving stability under drift.

10.2. Extension to Heterogeneous Architectures (CPU–GPU–FPGA)

Generalizing beyond CPUs requires expanding the feature schema and analytical priors to capture accelerator-specific phenomena: GPU SM occupancy, warp divergence, tensor-core utilization, HBM traffic patterns, PCIe/NVLink transfer costs, and FPGA pipeline initiation intervals. We will extend the hardware graph to include device–link–memory triplets and introduce cross-device contention features (e.g., shared PCIe switch congestion). Analytical baselines (roofline/ECM) will incorporate mixed-precision tensor ceilings and overlap models for compute–communication. Transfer learning can leverage meta-features describing kernel classes (dense GEMM vs. sparse SpMM) to enable few-shot adaptation when moving between vendors or new board revisions, targeting portable predictions and joint CPU–GPU–FPGA scheduling.

10.3. Online and Real-Time Prediction Models

While the current system supports fast inference, production environments benefit from truly online learners that update within seconds as workload mix and co-tenant pressure shift. We plan to add streaming stochastic updates (e.g., Bayesian last-layer, delta boosting) with strict compute budgets, coupled with drift detectors that trigger bounded recalibration without pausing service. For sub-millisecond control loops, we will investigate ultra-compact student models (e.g., pruned trees or linear–splines with monotonic constraints) that preserve calibration after distillation. Finally, we will expose multi-horizon forecasts (short, medium, long), enabling controllers to plan ahead for expected tail growth, power caps, or thermal conditions instead of reacting only to the present window.

10.4. Integration with Edge–Cloud Continuum

Edge deployments introduce heterogeneous hardware, intermittent connectivity, and stringent power envelopes. We will adapt the framework to federated telemetry and training, allowing sites to learn local counter–performance mappings while sharing only model updates or summary statistics for global robustness. Hierarchical models can split responsibilities: lightweight predictors at the edge provide rapid “good-enough” guidance, while cloud backends run heavier ablations and periodically distill refreshed students for device classes. Policy synchronization will consider bandwidth limits and privacy constraints, and incorporate geo-aware factors (ambient temperature, diurnal load). This edge–cloud continuum aims to deliver consistent, explainable performance optimization from embedded many-core devices to large data-center nodes.

11. Conclusion

This work introduced an AI-driven framework for predictive performance modeling on modern multi-core and many-core systems, unifying analytical structure (roofline/ECM/queueing) with data-centric learning over hardware counters, topology descriptors, and short temporal traces. By casting theory as a physics-informed baseline and learning only the residuals, the model preserves interpretability while capturing cross-layer effects NUMA locality, cache/coherence contention, branch irregularity, and phase behavior that defeat purely manual models. Calibrated uncertainty, SHAP-based attributions, and topology-aware reasoning (via a hardware graph) turn predictions into safe, actionable guidance for schedulers and autotuners. Empirically, the approach delivered low error with well-behaved prediction intervals across in-domain and leave-platform/workload splits, and translated forecasts into tangible efficiency gains in latency, energy, and EDP.

Beyond raw accuracy, the framework's value lies in operational impact: millisecond-class inference enables real-time "what-if" exploration, while online adaptation maintains calibration under drift and co-tenant interference. The modular design data acquisition, modeling engine, policy module, and orchestration supports portable deployment from HPC clusters to cloud and edge environments. Limitations remain: data heterogeneity, residual overheads, and fragility when leaping to novel microarchitectures or chiplet fabrics. Addressing these requires richer platform descriptors, few-shot adaptation, and tighter coupling with safe, model-based RL to close the loop between prediction and control. Extending the schema to heterogeneous CPU–GPU–FPGA stacks and federated edge–cloud settings is a natural path forward. Taken together, the results indicate that hybrid, uncertainty-aware analytics provide a practical foundation for explainable, portable performance optimization as core counts and system complexity continue to rise.

References

[1] Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*. https://crd.lbl.gov/assets/pubs_presos/roofline/roofline-CACM.pdf

[2] Stengel, H., Treibig, J., Hager, G., & Wellein, G. (2015). Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. *Proceedings of the 29th Int'l Conference on Supercomputing (ICS)*. https://arxiv.org/abs/1511.02136

[3] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., & von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation. *ACM SIGPLAN Notices*. https://dl.acm.org/doi/10.1145/165854.165874

[4] Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*. https://dl.acm.org/doi/10.1145/1465482.1465560

[5] Gustafson, J. L. (1988). Reevaluating Amdahl's Law. *Communications of the ACM*. https://dl.acm.org/doi/10.1145/42411.42415

[6] Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. *PACT*. https://parsec.cs.princeton.edu/parsec3-doc.htm

[7] SPEC. (2017). SPEC CPU2017 Benchmark Suite. *Standard Performance Evaluation Corporation*. https://www.spec.org/cpu2017/

[8] Bailey, D. H., Barszcz, E., Barton, J. T., et al. (1991). The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*. https://www.nas.nasa.gov/publications/npb.html

[9] Beamer, S., Asanović, K., & Patterson, D. (2015). The GAP Benchmark Suite. *arXiv preprint*. https://arxiv.org/abs/1508.03619

[10] McCalpin, J. D. (1995). STREAM: Sustainable Memory Bandwidth in High Performance Computers. *Technical Report*. https://www.cs.virginia.edu/stream/

[11] Mucci, P., Browne, S., Deane, C., & Ho, G. (1999/2008). PAPI: A Portable Interface to Hardware Performance Counters. *ICPP Workshops*. https://icl.utk.edu/papi/

[12] Weaver, V. (2013). Linux perf_event Features and Overhead. *FastPath '13*. https://perf.wiki.kernel.org/index.php/Main_Page

[13] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., & Le, C. (2010). RAPL: Memory Power Estimation and Capping. *Intel Technology Journal*. https://www.intel.com/content/www/us/en/developer/articles/technical/rapl-power-model.html

[14] Chen, T., Moreau, T., Jiang, Z., et al. (2018). TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *OSDI*. https://arxiv.org/abs/1802.04799

[15] Chen, T., Zheng, L., Yan, E., et al. (2018). Learning to Optimize Tensor Programs (AutoTVM). *NeurIPS*. https://arxiv.org/abs/1805.08166

[16] Zheng, L., Ye, S., Zhang, C., et al. (2020). Ansor: Generating High-Performance Tensor Programs for Deep Learning. *OSDI*. https://arxiv.org/abs/2006.06762

[17] Kaufman, S., Moskovitch, Y., Stephenson, M., Amarasinghe, S., & Yahav, E. (2019). Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation Using Deep Neural Networks. *ICML*. https://arxiv.org/abs/1808.07412

[18] Lundberg, S. M., & Lee, S.-I. (2017). A Unified Approach to Interpreting Model Predictions. *NeurIPS (SHAP)*. https://arxiv.org/abs/1705.07874

[19] Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. *NeurIPS*. https://papers.nips.cc/paper_files/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html

[20] Rasmussen, C. E., & Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. *MIT Press*. https://gaussianprocess.org/gpml/chapters/

[21] Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource Management with Deep Reinforcement Learning (DeepRM). *HotNets*. https://dl.acm.org/doi/10.1145/3005745.3005750

[22] Curtsinger, C., & Berger, E. D. (2013). STABILIZER: Statistically Sound Performance Evaluation. *ASPLOS*. https://dl.acm.org/doi/10.1145/2451116.2451141

[23] Viebke, A., Pllana, S., Memeti, S., & Kolodziej, J. (2019). Performance Modelling of Deep Learning on Intel Many Integrated Core Architectures. arXiv.

[24] Zhang, P., Fang, J., Yang, C., Huang, C., Tang, T., & Wang, Z. (2020). Optimizing Streaming Parallelism on Heterogeneous Many-Core Architectures: A Machine Learning Based Approach. arXiv.

[25] Hofmann, J., Alappat, C. L., Hager, G., Fey, D., & Wellein, G. (2019). Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors. arXiv.

[26] Butko, A., Bruguier, F., Novo, D., Gamatié, A., & Sassatelli, G. (2019). Exploration of Performance and Energy Trade-offs for Heterogeneous Multicore Architectures. arXiv.

[27] Arndt, O. J., Lüders, M., Riggers, C., et al. (2020). Multicore Performance Prediction with MPET: Using Scalability Characteristics for Statistical Cross-Architecture Prediction. Journal of Signal Processing Systems, 92, 981-998.

[28] Designing LTE-Based Network Infrastructure for Healthcare IoT Application - Varinder Kumar Sharma - IJAIDR Volume 10, Issue 2, July-December 2019. DOI 10.71097/IJAIDR.v10.i2.1540

[29] Krishna Chaitanaya Chittoor, "Architecting Scalable Ai Systems For Predictive Patient Risk", INTERNATIONAL JOURNAL OF CURRENT SCIENCE, 11(2), PP-86-94, 2021, https://rjpn.org/ijcspub/papers/IJCSP21B1012.pdf

[30] Thallam, N. S. T. (2020). Comparative Analysis of Data Warehousing Solutions: AWS Redshift vs. Snowflake vs. Google BigQuery. *European Journal of Advances in Engineering and Technology*, *7*(12), 133-141.

[31] The Role of Zero-Emission Telecom Infrastructure in Sustainable Network Modernization - Varinder Kumar Sharma - IJFMR Volume 2, Issue 5, September-October 2020. https://doi.org/10.36948/ijfmr.2020.v02i05.54991

[32] Thallam, N. S. T. (2021). Performance Optimization in Big Data Pipelines: Tuning EMR, Redshift, and Glue for Maximum Efficiency.