Check for updates

Concurrency and Computation: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

Execution Time Optimization Through Feature and Temporal Reduction in Asset Pricing

Umit Demirbaga[1] | Yue Xu[2] | Evrim Guler[3]

[1]Software Engineering, Ankara University, Ankara, Türkiye | [2]Business School, Durham University, Durham, UK | [3]Computer Engineering, Bartin University, Bartin, Türkiye

Correspondence: Umit Demirbaga (umitdemirbaga@gmail.com)

Received: 18 February 2026 | **Revised:** 6 June 2026 | **Accepted:** 17 July 2026

Keywords: artificial intelligence | cost-aware optimization | execution time | financial machine learning | scalability

ABSTRACT

High-dimensional financial machine learning (ML) pipelines are computational workloads as much as predictive models: their practical value depends on runtime, memory footprint, scalability, and the ability to retrain under resource constraints. This paper treats empirical asset pricing as a demanding real-world workload and proposes *Cost-Aware Adaptive Optimization*, using XGBoost (extreme gradient boosting), as a systems-aware framework to improve the computational efficiency of gradient-boosted tree pipelines while preserving application-level predictive utility. The framework uses computational feedback; optimization is driven by training time, prediction time, memory footprint, feature dimensionality, temporal training-window depth, model complexity, and marginal utility per unit of computation. We first profile 10 ML algorithms in a controlled high-performance computing setting and decompose execution cost into training and prediction components. XGBoost is then selected as the optimization target because it offers a strong cost–utility baseline on the asset-pricing workload. The proposed framework adaptively controls feature-dimensionality and temporal-depth choices, evaluates memory- and execution-aware trade-offs, and analyzes scalability through runtime growth, parallel speed-up, and parallel efficiency. Application-level utility measures, including out-of-sample R^2, mean squared error (MSE), and the Sharpe ratio of decile-sorted long–short portfolios, are retained as application-level validation metrics rather than as the sole objective. The results show that meaningful computational savings can be obtained by treating dimensionality, temporal depth, and model complexity as controllable workload variables. The study therefore contributes a computational feedback-driven optimization framework for scalable financial ML execution, framing workload adaptation and configuration selection as runtime-, memory-, and utility-aware system problems.

1 | Introduction

The scale of modern ML workloads continues to expand, and computing costs—once merely a minor implementation detail—have become a core issue in ML system design [1]. High-dimensional input spaces, long time-series historical data, repeated retraining cycles, and model complexity are the core factors that determine whether an ML pipeline can be successfully deployed [2]; while gradient boosting trees deliver excellent predictive performance, their training and memory costs surge sharply as the number of observations, features, and candidate configurations grows [3]. Therefore, model evaluation cannot rely solely on accuracy; execution time, memory footprint, scalability, and parallel efficiency must also be incorporated as core evaluation dimensions.

This paper studies these issues using empirical asset pricing as a high-dimensional computational workload. The financial prediction task is not presented as the paper's primary contribution; rather, it serves as a realistic stress test, with a

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2026 The Author(s). *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

data matrix containing many firm-level characteristics, a long temporal panel, repeated out-of-sample training cycles, and latency-sensitive prediction requirements. Previous asset-pricing studies have shown that ML methods can improve return prediction (e.g., [4–6]), but the associated computational pipeline is often evaluated primarily through statistical or investment performance measures. Such a view is incomplete in production-oriented, high-performance settings, where a model that offers only marginally higher accuracy may be unattractive if it requires excessive runtime, memory, or parallel resources.

We therefore propose *Cost-Aware Adaptive Optimization*, which uses XGBoost, as a computational-feedback-driven framework for scalable financial ML pipelines. The framework treats feature-dimensionality, temporal training-window depth, and model complexity as controllable workload variables. It uses computational telemetry – including training time, prediction time, memory footprint, dimensionality, temporal depth, and marginal utility per unit computation –to guide workload adaptation and identify configurations that provide favorable cost–utility trade-offs. Application-level utility outputs, such as out-of-sample R^2, MSE, and long-short portfolio Sharpe ratios, are retained as application-level validation evidence rather than as the sole optimization target.

The proposed framework is deliberately different from standard feature selection [7], ordinary temporal truncation [8], and conventional hyperparameter tuning [9]. Standard feature selection typically aims to improve statistical accuracy, reduce noise, or support interpretability [10]. In contrast, our feature-dimensionality control is formulated as a systems-level reduction of workload size under runtime and memory constraints. Standard temporal truncation is often used as a robustness exercise for sample periods [11]; here, temporal-depth control is used to determine whether older observations justify their computational cost. Conventional hyperparameter tuning ranks configurations mainly by predictive loss; our cost-aware optimization ranks configurations by retained utility relative to execution time, memory footprint, and model complexity [12].

We first benchmark 10 ML algorithms on the asset-pricing workload to characterize the baseline cost–utility landscape. XGBoost is then selected as the main optimization target because it offers a strong combination of predictive utility and computational feasibility compared with more expensive nonlinear alternatives and less accurate linear models [13]. The evaluation combines baseline predictive comparisons, training-time profiles, memory-footprint measurements, scalability curves, parallel speed-up and efficiency analyses, temporal-depth behavior, ablation tests, and cost–utility trade-off analysis.

The paper makes the following contributions. First, it provides a runtime-aware benchmark of 10 ML algorithms on a high-dimensional financial ML workload, decomposing execution cost into training and prediction components. Second, it introduces *Cost-Aware Adaptive* optimization, a computational feedback-driven framework for selecting efficient XGBoost configurations under cost and utility constraints. Third, it formulates feature-dimensionality and temporal-depth control as workload-adaptation mechanisms rather than finance-specific feature-selection or sample-period exercises. Fourth, it extends model evaluation beyond accuracy by incorporating memory footprint, execution time, and cost-normalized utility. Fifth, it analyses scalability, parallel speed-up, and parallel efficiency to identify systems-level bottlenecks and limits to concurrency. Sixth, it formulates configuration selection as a cost-normalized workload optimization problem at runtime, subject to memory, scalability, and retained-utility constraints.

The rest of this paper is organized as follows. Section 2 positions the paper within scalable ML, runtime-aware evaluation, memory-aware execution, high-performance computing (HPC) benchmarking, and cost-aware workload optimization. Section 3 motivates the asset-pricing task as a resource-intensive computational workload and defines the relevant cost dimensions. Section 4 describes the data, baseline ML pipeline, and the proposed Cost-Aware Adaptive optimization framework. Section 5 reports application-level predictive validation together with execution-time, memory-footprint, scalability, parallel-efficiency, temporal-depth, and ablation results. Section 6 discusses the systems-level implications and limitations. Finally, Section 7 concludes by outlining the implications for scalable, systems-aware financial ML pipelines.

2 | Related Work and Computational Positioning

This section positions the study across scalable ML systems, budget-aware model selection, boosted-tree computation, workload reduction, HPC benchmarking, financial ML validation, and model interpretation. Table 1 contrasts these streams with the present study by summarizing their technical contributions, primary workload variables, and relationship to the proposed framework. The comparison shows that prior work typically addresses one dimension of the problem–such as scalable training, hyperparameter search, predictive validation, or interpretability–whereas this study jointly integrates workload reduction, systems telemetry, scalability analysis, and financial validation into a cost-normalized framework for repeated, resource-constrained ML execution.

2.1 | Scalable Machine Learning and Runtime-Aware Model Evaluation

Traditional ML evaluation is frequently organized around predictive accuracy, error minimisation, or task-specific utility. These measures remain necessary, but they are insufficient when the model is part of a repeated computational pipeline. Scalable ML research treats training cost, prediction latency, resource consumption, repeatability, and deployment feasibility as first-class evaluation criteria rather than secondary implementation details. Large-scale learning systems must therefore be assessed not only by whether they improve an accuracy metric, but also by whether they can be trained, validated, and redeployed within realistic compute budgets [14–16]. HPC-oriented distributed deep-learning studies show that software stack choices, communication libraries, and cluster execution strategies materially affect training throughput and productivity [17], while workflow-oriented studies show that real-time ML and

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 | Technical positioning of related work and the present study.

Stream or study	Technical contribution	Main variables	Position relative to this paper
Scalable ML and workflows [14–18]	Large-scale SGD, parameter-server learning, distributed runtimes, and real-time ML workflows.	Throughput; communication cost; latency; retraining budget.	Establishes the systems view of ML execution.
Budget-aware model selection [19–22]	Random search, Bayesian optimization, successive halving, and Hyperband-style early stopping.	Search budget; validation cost; pruning; stopping rules.	Optimizes candidate search, but not the full repeated workload.
Boosted-tree systems [3, 23–26]	Gradient boosting, XGBoost, LightGBM, CatBoost, and comparative boosted-tree evidence.	Features; samples; tree depth; boosting rounds; split search; parallelism.	Provides the modeling substrate targeted for workload optimization.
Feature and temporal adaptation [27–30]	Feature selection, adaptive windows, and concept-drift adaptation in changing data.	Retained features; window length; relevance decay; non-stationarity.	Usually targets accuracy, interpretability, or drift rather than execution cost.
HPC and performance measurement [31–37]	Grid/HPC benchmarking, task scheduling, cloud cost–performance, speed-up theory, and MLPerf measurement.	Memory footprint; cores; speed-up; parallel efficiency; instance cost.	Supplies evaluation metrics, but not a domain-specific adaptive pipeline.
Financial ML validation [4–6, 38–43]	ML return prediction, nonlinear characteristic models, shrinkage, and open asset-pricing libraries.	Out-of-sample R^2; MSE; Sharpe ratio; firm characteristics.	Emphasizes prediction and economic utility more than execution efficiency.
Model interpretation [44, 45]	Local surrogate explanations and Shapley-value attribution.	Feature contribution; local fidelity; attribution stability.	Explains model behavior rather than controlling runtime and memory.
Present study	Cost-aware adaptive XGBoost optimization for repeated financial ML execution.	Training and prediction time; memory footprint; feature count; temporal depth; model complexity; utility retention; speed-up and efficiency.	Integrates workload reduction, systems telemetry, and financial validation into a single cost-normalized execution framework.

visualization pipelines must be designed around end-to-end execution constraints rather than model accuracy alone [18].

This distinction is particularly important for workloads involving rolling or repeated retraining. A one-off accuracy benchmark may hide the cumulative cost of repeatedly rebuilding feature matrices, fitting models, validating configurations, and generating predictions. Runtime-aware evaluation is therefore closer to a systems benchmark than to a conventional statistical comparison: it must measure execution time, variability across runs, prediction latency, and the pipeline's sensitivity to workload size. Hyperparameter- and resource-aware optimization studies similarly show that model selection should account for the computational budget, random-search baselines, surrogate modeling, and early stopping, rather than treating all candidate configurations as equally affordable [19–22]. The present study follows this systems-oriented view by evaluating financial ML as a repeated workload whose utility must be interpreted jointly with computational cost.

2.2 | Computational Characteristics of Gradient-Boosted Tree Models

Gradient-boosted tree models are attractive for high-dimensional tabular workloads because they often achieve strong predictive performance while remaining efficient. Their methodological foundation follows gradient boosting as functional gradient descent over additive tree ensembles [23]. XGBoost is a prominent scalable implementation: it introduced a regularized tree-boosting system with sparsity-aware split finding, weighted quantile sketching, cache-aware computation, and support for parallel and distributed execution [24]. Later systems such as LightGBM and CatBoost further emphasize histogram-based learning, leaf-wise growth, handling of categorical features, and unbiased boosting strategies for efficient tabular prediction [25, 26]. Comparative reviews confirm that the empirical strength of boosting comes with implementation-dependent computational trade-offs rather than cost-free accuracy improvements [3].

These characteristics make XGBoost a natural candidate for production-oriented ML pipelines that must balance predictive utility and computational feasibility. Several workload and model parameters nevertheless shape XGBoost's computational cost. Training costs depend on the number of observations, the number of features, the tree depth, the number of boosting rounds, the split-search procedure, the subsampling choices, and the parallel execution strategy. The related literature, therefore, motivates evaluating boosted trees not only as statistical predictors but also as configurable computational workloads whose runtime and memory behavior respond to controllable design choices.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2.3 | Feature Dimensionality, Temporal Depth, and Workload Reduction

In a high-dimensional tabular pipeline, the number of retained features affects data loading, matrix construction, split evaluation, memory access, and prediction traversal, while in a long-panel setting, the temporal training window length determines how many observations are included in each retraining cycle. Both quantities, therefore, influence runtime, memory footprint, and the cost of repeated execution. Classical feature-selection research frames dimensionality reduction as a way to remove irrelevant variables, reduce variance, improve generalization, or support interpretation [27, 28]. In streaming and non-stationary settings, adaptive windows and concept-drift studies show that older observations may become less useful when the data-generating process changes, motivating explicit control over the temporal span used for learning [29, 30].

This paper treats feature count and temporal depth as computational variables rather than interpretability devices. Standard feature selection usually aims to improve statistical performance, reduce noise, or produce a more interpretable model. Standard temporal truncation is often used to test robustness across sample periods or to address non-stationarity. These approaches are useful but incomplete for a systems paper unless they are embedded in a cost-aware optimization framework. A reduced feature set or shorter training window is computationally valuable only if it preserves sufficient retained utility while reducing execution time, memory footprint, or retraining cost. Therefore, the proposed framework evaluates dimensionality and temporal depth through cost-normalized utility rather than through a finance-specific interpretation.

2.4 | Memory Footprint, Parallel Efficiency, and HPC-Based ML Benchmarking

Reporting execution time alone is not sufficient for systems-level evaluation. Memory footprint determines whether a workload fits within a target machine, whether multiple runs can be executed concurrently, and whether parallel execution creates resource contention. Peak memory is especially important for tree-based models because intermediate data structures, split statistics, and repeated training windows can increase memory pressure even when prediction latency remains small. HPC-oriented benchmarking research has long emphasized that benchmark design, middleware, scheduling, and platform selection determine whether a parallel workload is practically deployable [31]. Task-based runtime systems such as StarPU further illustrate that scheduling and heterogeneous-resource management are central to performance, not incidental implementation choices [32]. Recent cloud-HPC cost studies extend this view to AWS-based execution, where performance must be interpreted together with monetary cost and instance selection [33].

Parallel speed-up and parallel efficiency provide additional information about scalability. A workload may become faster when more central processing unit (CPU) cores are allocated. Still, sublinear speed-up can arise from synchronization, scheduling overhead, memory-bandwidth limits, serial sections of the algorithm, and workload imbalance. Classical parallel-performance analysis shows that speed-up is bounded by the non-parallelizable component of a computation [34], while scaled-speed-up arguments explain why larger workloads may benefit differently from added processors [35]. Modern ML benchmarking efforts similarly emphasize reproducible system-level measurement, including training time, hardware use, scaling behavior, and implementation constraints [36, 37]. For this reason, the present paper reports memory footprint, runtime scalability, parallel speed-up, and parallel efficiency in addition to training and prediction time. These measures are needed to determine whether adaptive optimization improves the pipeline's deployability rather than merely changing a single timing parameter.

2.5 | Positioning Against Finance-Oriented and Interpretation-Oriented Studies

The paper is related to empirical financial ML studies only insofar as asset pricing provides the validation data. Prior work in this area has primarily evaluated predictive performance, return predictability, or portfolio-level utility [4–6, 38, 39]. Related asset-pricing studies also use nonlinear or high-dimensional methods to model characteristic interactions, construct return cross-sections, or shrink large sets of predictors [40–42]. Open-source characteristic libraries further demonstrate the scale and reproducibility challenges created by hundreds of return predictors [43]. Still, they do not make computational cost, memory footprint, parallel efficiency, or repeated workload execution the central methodological contribution. This study differs by treating the asset-pricing pipeline as a high-dimensional systems workload and by making cost-aware optimization the main object of study.

The paper is also separate from interpretation-oriented analyses whose primary objective is to interpret model behavior or rank predictors for substantive interpretation. In the present framework, however, feature thresholds, temporal-window lengths, model-complexity settings, and stopping rules are treated as workload-control variables. Its novelty lies in computational feedback-driven adaptation: configurations are selected based on runtime, prediction latency, memory footprint, feature dimensionality, temporal depth, model complexity, and marginal utility per unit of computation.

3 | Computational Motivation and Workload Characteristics

Empirical asset pricing provides a demanding test case for computationally efficient ML because the workload combines high dimensionality, long temporal depth, repeated model execution, and latency-sensitive prediction. The objective of this paper is not to make a finance-specific claim about the economic interpretation of individual predictors, but to use this setting as a realistic workload on which cost-aware optimization can be evaluated. The data matrix is large in both directions: each monthly cross-section contains many firm-level characteristics, and the historical panel spans several decades, providing a useful stress test for algorithms' runtime and memory behavior.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The first core computational challenge faced by ML models trained on hundreds of enterprise-level features is the high feature dimensionality. These models must repeatedly process high-dimensional feature matrices, and tree-based methods such as XGBoost incur a sharp increase in computational and storage costs as the scale of features grows. Feature-dimensionality is therefore not only a statistical modeling issue; it is a controllable workload parameter that directly affects training time, prediction latency, cache behavior, and memory use. A systems-aware pipeline should be able to decide whether the marginal utility of additional features justifies their computational burden.

The second challenge is temporal depth. Long historical panels increase the number of training observations and can raise both runtime and memory pressure. In rolling out-of-sample evaluation, the training window is repeatedly reconstructed or updated, and the model is retrained across multiple evaluation periods. A longer temporal window may provide more information, but it also increases data loading, preprocessing, and fitting time, as well as peak memory requirements. Temporal window length is therefore treated in this study as a training-depth control variable rather than merely as a finance-motivated sample-period choice.

The third challenge is the repeated execution of workloads. Out-of-sample financial ML pipelines are rarely executed once. They typically require repeated training, validation, prediction, and portfolio-evaluation steps across rolling or expanding windows. This repetition amplifies even moderate inefficiencies: a small increase in per-run training time or memory footprint can become substantial when multiplied across algorithms, configurations, windows, and repeated runs. For this reason, the paper separates one-off model quality from the cost of repeated pipeline execution.

Training time and prediction time are measured separately because they represent different computational costs. Training time captures the cost of fitting the model, including tree construction, evaluation of candidate splits, and optimization of model parameters. Prediction time captures the cost of applying the trained model to new observations, which is relevant when forecasts must be generated repeatedly after each retraining cycle. A configuration may be acceptable for offline training but unsuitable for frequent deployment if prediction latency is excessive; conversely, a model with fast prediction may still be impractical if retraining is too expensive.

Memory footprint is an additional deployment and scalability constraint. Peak memory footprint determines whether a workload can be executed on a given machine, how many configurations can be evaluated concurrently, and whether parallel execution causes resource contention. Memory pressure is especially relevant when feature-dimensionality, temporal-window, and model-complexity controls are considered together. A cost-aware optimization framework must therefore evaluate memory footprint alongside runtime and predictive utility.

Parallel execution and CPU utilization are also central to the systems-level interpretation of the workload. XGBoost and related tree-based methods can exploit multi-core computation, but speedups are not necessarily linear due to synchronization costs, memory bandwidth limits, scheduling overheads, and workload imbalance. Parallel speed-up and parallel efficiency, therefore, provide information that raw runtime cannot capture. They indicate whether additional cores are being used effectively and whether the optimized workload scales under realistic compute constraints.

4 | Computational Framework and Experimental Protocol

The methodological objective is to optimize a high-dimensional financial ML pipeline as a computational workload rather than merely to compare forecasting models. We model one-month-ahead stock excess returns using supervised learning, but the central design problem is systems-oriented: for a given predictive workload, how can execution time, memory footprint, temporal training depth, feature-dimensionality, model complexity, and parallel efficiency be controlled while retaining adequate application-level utility? Let $R_{i,t+1}$ be the excess return of stock i at date $t+1$, and let $\boldsymbol{z}_{i,t} \in \mathbb{R}^N$ denote the vector of predictive characteristics, where $N = 209$. The underlying prediction function is estimated through a rolling out-of-sample pipeline, but model configurations are assessed using both application metrics and computational telemetry.

The proposed *Cost-Aware Adaptive Optimization* framework proceeds in three stages. First, we profile a benchmark set of linear and nonlinear ML algorithms to establish the baseline runtime–utility landscape of the workload. Training time, prediction time, memory footprint, and repeated-run variability are recorded in a controlled high-performance computing environment, while out-of-sample R^2, MSE, and long-short portfolio Sharpe ratios are used only as application-level validation measures. Second, XGBoost is selected as the optimization target because it provides a strong baseline trade-off between predictive utility and computational feasibility. Third, the framework adaptively adjusts feature dimensionality and temporal window depth and evaluates each configuration based on marginal utility per unit of computation rather than predictive accuracy alone.

This formulation intentionally differs from conventional feature selection, temporal truncation, and hyperparameter tuning. Feature-dimensionality control is treated as workload-size adaptation under runtime and memory constraints rather than as finance-factor interpretation. Temporal-window control is treated as a mechanism for regulating training depth and computational load, not simply as a sample-period robustness check. Model-complexity choices are evaluated by cost-normalized utility rather than only by validation loss.

Figure 1 depicts the proposed system-oriented framework for runtime, resource, and utility-aware execution.

4.1 | Benchmark Model Set and Computational Profiles

The benchmark model set is selected to span distinct algorithmic families and computational profiles, rather than to provide only a predictive comparison. Linear, ridge, and lasso regression are included as low-complexity baselines with comparatively small

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

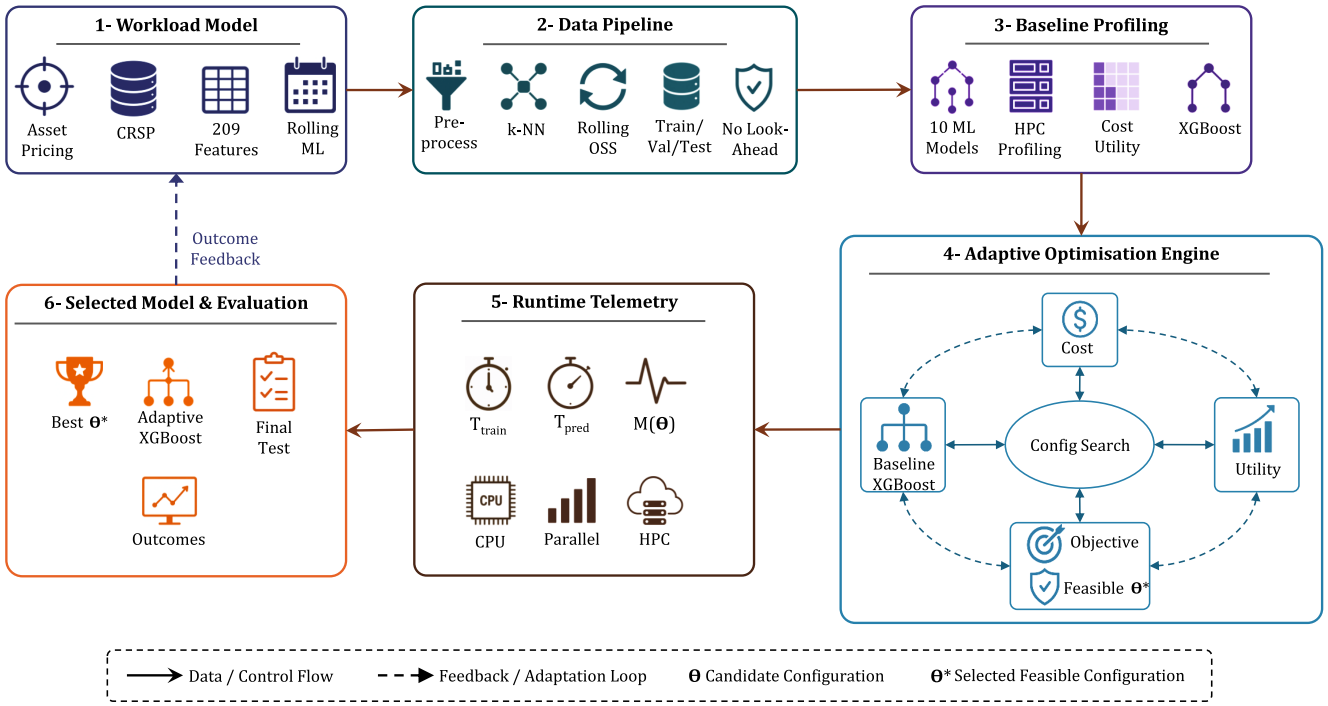


FIGURE 1 | Cost-aware adaptive optimization for scalable financial ML.

training and memory requirements. A decision tree is included as a single-tree nonlinear baseline whose cost is determined by recursive split search, without ensemble aggregation. Random forests are bagging-based ensembles with substantial parallelization potential. However, this advantage can come at the cost of higher memory use and training time, as many trees are stored and evaluated. Gradient boosting is included as a sequential boosting baseline in which later trees depend on earlier residual structure, limiting some forms of parallelism.

XGBoost is included as an optimized gradient-boosted tree implementation with strong expected cost–utility behavior on high-dimensional tabular data. Support vector machines remain high-cost nonlinear benchmarks whose training can become expensive as the number of observations increases. K-nearest neighbors is included as a lazy-learning baseline: it has low model-fitting cost but can incur high prediction cost because predictions require distance computations against stored observations. Neural network configurations are iterative, optimization-based models whose computational behavior depends on the architecture, the number of parameters, the number of training iterations, and the convergence properties. This model set therefore supports workload profiling across low-complexity statistical baselines, single-tree methods, ensemble methods, kernel-based learning, lazy learning, and iterative neural optimization.

4.2 | Cost-Aware Adaptive Optimization

We define the financial ML pipeline as a repeated supervised-learning workload

$$\mathcal{W} = \left\{ \mathcal{D}_{train}^{(r)}, \mathcal{D}_{val}^{(r)}, \mathcal{D}_{test}^{(r)} \right\}_{r=1}^{R}, \tag{1}$$

where r indexes rolling out-of-sample executions. Each execution trains an XGBoost model on a selected feature subset and temporal window, validates the configuration, and generates predictions for the next evaluation period. The optimization problem is to select a configuration that preserves sufficient predictive and application-level utility while reducing execution cost and memory footprint.

Let $\theta = (\tau, L, \psi, s)$ denote a candidate configuration. The decision variable τ is the feature-dimensionality threshold used to determine the retained feature set $\mathcal{F}_\tau \subseteq \mathcal{F}$; L is the temporal training-window length; ψ denotes XGBoost model-complexity parameters such as the number of estimators, maximum depth, learning rate, subsampling rate, and column-sampling rate; and s is a computational stopping threshold used to terminate the search when additional complexity no longer provides sufficient marginal benefit. They define a computational search grid over progressively smaller feature sets, allowing the framework to trace the cost–utility response as feature-dimensionality decreases. For a configuration θ, predictive utility is represented by out-of-sample $R^2(\theta)$ and $MSE(\theta)$. In contrast, downstream application-level utility is represented by the validation-portfolio Sharpe ratio, denoted $SR(\theta)$. These terms are used as application-level retained-utility measures in the cost-normalized objective.

The computational cost of configuration θ is defined as.

$$C(\theta) = \alpha T_{train}(\theta) + \beta T_{pred}(\theta) + \gamma M(\theta) + \delta \Omega(\psi), \tag{2}$$

where $T_{train}(\theta)$ is training time, $T_{pred}(\theta)$ is prediction time, $M(\theta)$ is peak memory footprint, and $\Omega(\psi)$ is a model-complexity penalty. The non-negative weights α, β, γ, and δ allow the cost function to represent different deployment priorities, such as faster retraining, lower prediction latency, or reduced memory pressure.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The retained utility of a configuration is measured relative to a baseline XGBoost model, θ_0, trained on the full feature set and the reference temporal window. We define a combined utility score as.

$$U(\theta) = \lambda_1 R^2(\theta) - \lambda_2 MSE(\theta) + \lambda_3 SR(\theta), \tag{3}$$

where λ_1, λ_2, and λ_3 control the relative importance of predictive fit, prediction error, and downstream application utility. The cost-normalized optimization objective is then.

$$\theta^* = \arg\max_{\theta \in \Theta} \frac{U(\theta)}{C(\theta) + \varepsilon}, \tag{4}$$

subject to

$$\frac{U(\theta)}{U(\theta_0)} \geq \rho, \quad T_{train}(\theta) + T_{pred}(\theta) \leq B_T, \quad M(\theta) \leq B_M, \tag{5}$$

where ρ is the minimum retained-utility threshold, B_T is the maximum runtime budget, B_M is the maximum memory budget, and ε is a small positive constant for numerical stability.

A marginal-utility-per-computation rule can also be used to stop adding to the workload's complexity. For two successive configurations θ_k and θ_{k+1} that increase feature-dimensionality, temporal depth, or model complexity, the marginal gain is.

$$\Delta_{k+1} = \frac{U(\theta_{k+1}) - U(\theta_k)}{C(\theta_{k+1}) - C(\theta_k) + \varepsilon}. \tag{6}$$

In the empirical implementation, all cost and utility components are normalized before computing the cost-aware score. We use a deployment-neutral equal-weight setting, so the objective does not privilege a single metric such as accuracy, runtime, prediction latency, or memory use. The search can stop adding complexity when $\Delta_{k+1} < s$, indicating that the retained utility gain does not justify the additional computational cost.

Algorithm 1 summarizes the cost-aware selection mechanism used in the proposed framework. The procedure first fits a reference XGBoost model on the full workload to establish a baseline for utility and computational cost. It then evaluates candidate workloads formed by combining feature thresholds, temporal-window lengths, and XGBoost configurations. Each candidate is retained only if it preserves the required share of baseline utility and satisfies the runtime and memory budgets. Among the retained candidates, the algorithm selects the configuration with the highest utility per unit of computational cost. The selected model is therefore not necessarily the most complex or the most accurate in isolation; it is the feasible configuration that provides the best cost–utility trade-off for repeated financial ML execution.

ALGORITHM 1 | Cost-aware adaptive XGBoost Optimization.

```
Input: Training and validation data;
       feature thresholds 𝒯;
       temporal windows ℒ;
       XGBoost configurations Ψ;
       runtime budget B_T;
       memory budget B_M;
       utility threshold ρ.
Output: the best feasible configuration θ*
Fit baseline model θ_0 using the full feature set
  and reference window
Compute baseline utility U(θ_0) and cost C(θ_0)
Initialise feasible set 𝒜 ← ∅
for all(τ, L, ψ) ∈ 𝒯 × ℒ × Ψ
  Build candidate θ = (𝒯_τ, L, ψ)
  Train and validate θ on the rolling workload
  Measure U(θ), T(θ), M(θ), and C(θ)
  if θ satisfies the utility, runtime, and
   memory constraints then
    Add θ to 𝒜 with its cost-normalized score
      end if
end for
if 𝒜 = ∅ then
  return baseline configuration θ_0
end if
Select θ* with the highest cost-normalized
  score
return θ*, U(θ*), T(θ*), and M(θ*)
```

4.3 | Financial Prediction Task

The prediction task follows a supervised learning formulation for one-month-ahead excess stock returns. For stock i at month t, let $R_{i,t+1}$ denote the next-month excess return and let $z_{i,t}$ denote the vector of firm characteristics observed at time t. The prediction model is written as

$$R_{i,t+1} = E_t(R_{i,t+1}) + \epsilon_{i,t+1}, \tag{7}$$

$$E_t(R_{i,t+1}) = g^*(z_{i,t}). \tag{8}$$

Within the present study, this formulation defines the prediction target for the computational workload. The vector $z_{i,t}$ supplies the high-dimensional input matrix on which feature-dimensionality control operates, while the rolling estimation of $g^*(.)$ supplies the repeated training and prediction cycles used to evaluate runtime, memory footprint, and retained utility.

4.4 | Rolling Out-Of-Sample Design and Validation Logic

The empirical workload is evaluated through a rolling out-of-sample design. For each rolling execution r, the historical data available up to the relevant decision date is split into training and validation windows, and the following test window is held out until the final evaluation step. The training window is used for preprocessing, imputation fitting, model fitting, and candidate model construction. The validation window is used to compare candidate configurations, compute retained-utility measures, and apply the cost-aware selection rule. The test window is used only after a configuration has been selected and provides the final out-of-sample predictive and application-level evaluation for that rolling execution.

All preprocessing, imputation, feature screening, temporal-window selection, model fitting, and hyperparameter or cost-aware configuration decisions are performed using only information available within the training and validation

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

windows. The test window is not used to select features, tune parameters, choose temporal depth, construct imputation rules, estimate feature thresholds, or decide the adaptive XGBoost configuration. This design makes the repeated pipeline a realistic computational workload and avoids mixing model-selection information with final evaluation information.

4.4.1 | Look-Ahead Bias Control

The pipeline is designed to avoid look-ahead bias. No future returns, future feature values, full-sample feature rankings, full-sample imputations, or test-period information are used during model training, feature-dimensionality control, temporal-window selection, or adaptive XGBoost optimization. In each rolling execution, imputation rules, feature thresholds, temporal-window choices, and model-complexity decisions are estimated using only the information available at that point in the rolling training and validation sequence.

4.5 | Measurement of Execution Time

Execution time is the wall-clock time required to complete the relevant stages of the ML pipeline for a given workload configuration. We report training time and prediction time separately because they correspond to different operational bottlenecks. Training time measures the cost of fitting the model during each rolling execution, whereas prediction time measures the cost of applying the fitted model to the evaluation set. This separation allows the analysis to distinguish configurations that are expensive to retrain from those that are expensive to deploy.

We examine 10 ML algorithms on the stock-level panel workload and run each execution five times to quantify runtime variability. The repeated-run protocol is used not only for statistical stability but also to assess the consistency of computational behavior under the same workload. Where relevant, the later scalability experiments extend this timing protocol by considering memory footprint, runtime growth, parallel speed-up, and parallel efficiency.

4.6 | Data

The monthly equity returns are obtained from the Centre for Research in Security Prices (CRSP) database {https://www.crsp.org/}. We include all U.S. firms listed on the New York Stock Exchange (NYSE), American Stock Exchange (AMEX), and Nasdaq Stock Market (NASDAQ). The sample period spans March 1957 to December 2022, providing a long temporal panel for repeated training-window construction. We use the one-month Treasury bill rate from CRSP as a proxy for the risk-free rate and compute equity excess returns accordingly.

The input matrix contains 209 firm-level predictive characteristics,[1] which define the high-dimensional feature space of the computational workload and provide the basis for evaluating feature-dimensionality control as a runtime and memory optimization mechanism [43].

Missing values are handled before model fitting because incomplete observations affect both predictive validity and computational execution. We use k-nearest neighbors (k-NN) imputation as the main strategy because it preserves local structure in the high-dimensional financial panel by estimating missing values from neighboring observations rather than replacing them with a global location statistic. The method is, however, computationally more expensive than mean or median imputation because it requires distance calculations in the feature space and can incur non-trivial preprocessing costs when applied repeatedly across rolling windows.

Imputation is treated as part of the controlled preprocessing pipeline. To avoid information leakage, imputation rules are fitted within the training window of each rolling execution and then applied to the corresponding validation and test data using only information available at that point in the sequence. The reported model-execution times focus on fitting and prediction after preprocessing, as described in the runtime protocol; Figure 7 reports the preprocessing cost for imputation separately within the sensitivity analysis.

Alternative imputation strategies are evaluated to separate the predictive effect of missing-value treatment from the computational effect of repeated preprocessing. The imputation-sensitivity experiment was conducted under the same rolling no-look-ahead protocol and is reported in Section 5.3 and Figure 7. The comparison evaluates mean, median, k-nearest neighbors, and MICE imputation in terms of retained predictive utility, application-level utility, and preprocessing cost.

4.7 | Experimental Setup and Reproducibility Protocol

The experimental protocol is designed to support a computing-oriented evaluation of the proposed framework. In addition to predictive and application-level utility, the protocol records runtime, prediction time, memory footprint, CPU-utilization records, parallel scaling behavior, repeated-run variability, and configuration-level reproducibility. The recorded logs provide the basis for the systems-level results reported later; derived quantities such as speed-up, parallel efficiency, and cost-normalized summaries are computed from these measurements.

4.7.1 | Hardware and Cloud Configuration

All experiments are executed in an Amazon Web Services (AWS)-based HPC environment configured with 64 CPU cores and 1 terabyte (TB) of main memory. This large-memory setting allows the peak memory footprint to be observed without prematurely terminating larger baseline configurations.

The hardware platform is kept fixed to isolate the effect of workload design and CPU-core allocation on execution behavior. The study does not claim to benchmark across different AWS instance families, processor architectures, GPU accelerators, or memory-capacity classes. The core-scaling experiment should instead be interpreted as a controlled resource-configuration benchmark within the same HPC environment, which avoids

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

confounding hardware-class effects with the algorithmic and workload-level effects studied in this paper.

4.7.2 | Software Environment and Implementation Details

All models are implemented in Python using standard ML and scientific-computing libraries, including NumPy, pandas, scikit-learn, and XGBoost. Software versions and operating-system details are fixed across experimental runs and recorded as part of the reproducibility protocol. Linear models, tree-based models, support vector machines, k-nearest neighbors, and neural network baselines are implemented using the corresponding standard library interfaces. XGBoost configurations are executed through the Python XGBoost interface, with thread-level parallelism controlled through the relevant core-count parameter during the scaling experiments.

4.7.3 | Runtime Measurement Protocol

Runtime is measured using wall-clock time for each model run. Training time, denoted T_{train}, is the elapsed time required to fit the model on the selected training window and feature set. Prediction time, denoted T_{pred}, is the elapsed time required to generate predictions on the evaluation data after the model has been fitted. Total execution time is defined as

$$T_{exec} = T_{train} + T_{pred}. \tag{9}$$

The timing protocol focuses on model execution after preprocessing. It includes model fitting and prediction but excludes raw data acquisition and data download.

4.7.4 | Memory-Footprint Measurement Protocol

Peak memory footprint is defined as the maximum memory consumed during model execution for a given configuration. Memory footprint is recorded during model execution because runtime alone does not determine whether a workload is deployable or scalable. Memory is therefore evaluated alongside runtime for the full XGBoost, tuned XGBoost, and adaptive XGBoost configurations, and the reported memory-footprint values in Section 5 refer to recorded peak measurements.

4.7.5 | Parallel Execution and Core-Scaling Protocol

The core-scaling experiment evaluates the runtime of full XGBoost and adaptive XGBoost across controlled CPU-core configurations on a fixed AWS-based HPC platform. The tested core counts are $p \in \{1, 2, 4, 8, 16, 32, 64\}$, within the available 64-core allocation. For each value of p, the corresponding XGBoost thread parameter is fixed to the selected core count, and the same workload configuration is executed under the repeated-run protocol. The experiment therefore varies only the CPU cores allocated to XGBoost, not the AWS instance family, processor architecture, accelerator type, or memory-capacity class. Parallel speed-up is defined as

$$S(p) = \frac{T(1)}{T(p)}, \tag{10}$$

where $T(1)$ is the single-core runtime and $T(p)$ is the runtime using p CPU cores. Parallel efficiency is defined as

$$E(p) = \frac{S(p)}{p}. \tag{11}$$

These measures distinguish absolute runtime improvement from the efficiency with which additional CPU resources are used. The reported speed-up and efficiency values are derived from the recorded wall-clock runtime logs rather than being measured separately.

4.7.6 | Repeated Execution and Statistical Stability

Each experiment is repeated five times to reduce the influence of transient runtime variation in shared or cloud-based computing environments, including background system activity, CPU scheduling, and memory management. This methodology is applied to the baseline algorithm comparison, feature-dimensionality experiments, temporal-window experiments, and adaptive XGBoost evaluations.

4.7.7 | Reproducibility and Configuration Control

To support reproducibility, random seeds, model configurations, feature-dimensionality thresholds, temporal-window lengths, CPU-core settings, runtime-measurement scripts, software-version records, operating-system details, and AWS instance metadata are fixed and documented for each experimental run. The same preprocessed data matrix is used when comparing configurations so that timing differences reflect model and workload changes rather than changes in raw input construction.

In addition, the reproducibility package for this study includes seven categories of technical material: experimental configuration files; random-seed and model-configuration records; feature-threshold and temporal-window definitions; CPU-core scaling settings; runtime and memory-measurement scripts; computational logs; and software, operating-system, and AWS metadata records. These materials support the replication of the reported computational experiments and clarify how each workload configuration was executed and measured.

The firm-level characteristic definitions used in the study are based on the Open Source Asset Pricing data library. However, the CRSP return data cannot be redistributed by the authors because they are subject to institutional data-license restrictions. Therefore, reproducibility is supported through documented configurations, computational logs, scripts, and publicly accessible characteristic definitions, while access to the licensed return data remains subject to the terms of use specified by the original data provider.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.8 | Computational Complexity Considerations

The computational cost of the workload depends on the number of observations n, the number of retained features d, the temporal-window length L, the number of candidate configurations $|\Theta|$, the XGBoost model-complexity parameters, the number of CPU cores p, and the peak memory footprint M. In a rolling design, these costs are multiplied across repeated executions, so even moderate per-run overhead can become material when many windows, feature thresholds, temporal lengths, and model configurations are evaluated.

At a high level, the repeated training burden for a rolling boosted-tree workload can be approximated as the product of the number of rolling executions, the number of candidate configurations evaluated, and the per-configuration XGBoost fitting cost. Let R denote the number of rolling executions, B the number of boosting rounds, D the maximum tree depth, n_L the number of observations induced by temporal-window length L, and d_τ the number of retained features under feature threshold τ. The repeated training burden can then be viewed qualitatively as increasing with

$$\mathcal{O}\big(R|\Theta| \cdot C_{\mathrm{XGB}}\big(n_L, d_\tau, B, D, p\big)\big), \tag{12}$$

where $C_{\mathrm{XGB}}(\cdot)$ denotes the implementation-dependent cost of fitting an XGBoost model. This expression is intended as an asymptotic-style workload description rather than a closed-form complexity result, because the exact cost depends on the implementation, split-search strategy, tree method, sparsity pattern, caching behavior, and parallel-execution details.

For histogram- or split-based tree construction, the fitting cost increases with the number of observations processed in each boosting round, the feature-dimensionality available for split search, the number of boosting rounds, and the depth of the trees. At a high level, larger n_L, d_τ, B, and D expand the effective split-search and tree-construction workload. The temporal window length L directly affects n_L, as longer windows include more observations in each rolling training cycle. The number of candidate configurations, $|\Theta|$, affects the outer optimization cost, as each candidate may require fitting and validation before it is accepted or discarded by the cost-aware objective.

Parallel execution can reduce the split-search and histogram-construction components of XGBoost training, but the reduction is not necessarily proportional to the number of cores. Synchronization, memory-bandwidth pressure, scheduling overhead, and non-parallel parts of the pipeline can limit speed-up. Hence, the effective runtime can be viewed qualitatively as a function $T(n, d, L, B, D, |\Theta|, p)$ in which increasing p may reduce runtime but may also reduce parallel efficiency. Peak memory footprint similarly increases with the size of the retained data matrix, the number of intermediate training structures, the number and depth of fitted trees, and the degree of concurrent execution.

This formalization clarifies how the proposed framework reduces the effective workload size. Feature-dimensionality control reduces d_τ and, consequently, the split-search and data-matrix burdens. Temporal-depth control reduces n_L by limiting the number of observations included in each rolling training window. The marginal utility-per-computation stopping rule prunes candidate configurations whose additional utility is too small relative to their computational cost. Adaptive configuration selection also avoids high-cost configurations that violate runtime or memory budgets. These mechanisms do not alter the underlying prediction task; they change the computational path used to reach an acceptable cost–utility configuration.

4.9 | Sensitivity and Robustness Checks

The robustness checks are designed to assess whether a single arbitrary workload setting drives the reported behavior. First, sensitivity to feature-threshold choices is evaluated through the feature-dimensionality experiments, which compare retained utility, execution time, and memory pressure across progressively reduced feature sets. These thresholds are workload-control points in a computational search grid, not interpretability claims or finance-specific factor-selection rules; their purpose is to approximate the cost-utility response and identify Pareto-relevant regions between retained utility, runtime, and memory footprint. Second, sensitivity to temporal-window length is evaluated through temporal-depth experiments that compare predictive utility and runtime across alternative training-window lengths. These checks align with the adaptive framework because feature count and temporal depth are treated as controllable computational variables.

Third, sensitivity to the imputation method is assessed by comparing the main k-NN strategy with lower-cost mean and median imputation, and with more flexible but computationally intensive MICE imputation, under the same rolling, no-look-ahead protocol. Fourth, sensitivity to CPU core count is assessed through core-scaling experiments, where full and adaptive XGBoost are evaluated at different values of p and compared using speed-up and parallel efficiency.

Fifth, repeated-run variability is addressed by executing each experiment five times and reporting mean values and standard deviations where applicable. Sixth, the ablation study isolates the contribution of the optimization components, including feature-dimensionality control, temporal-window control, and cost-aware stopping. These checks support interpreting the results as computational workload optimization rather than a finance-only asset-pricing comparison.

5 | Experimental Results and Systems-Level Evaluation

This section evaluates the proposed Cost-Aware Adaptive XGBoost optimization framework from a systems and computational-efficiency perspective. Empirical asset-pricing outputs are treated as application-level utility measures for a high-dimensional ML workload. The main evaluation criteria are runtime behavior, training and prediction cost, memory footprint, scalability, parallel efficiency, workload reduction, and cost-normalized utility.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

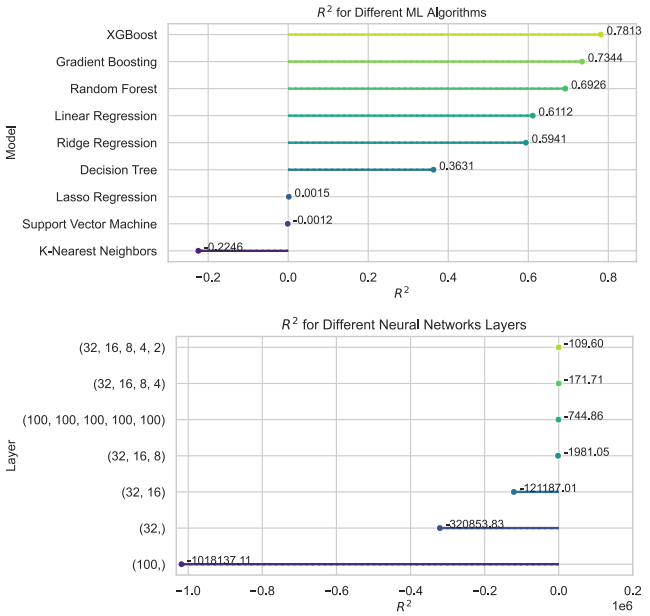


FIGURE 2 | Out-of-sample R^2 computed from repeated out-of-sample evaluations for the candidate algorithms.

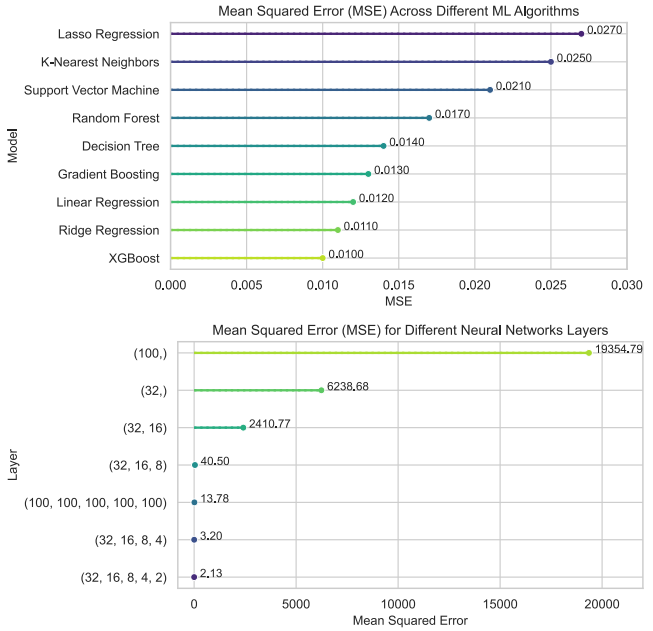


FIGURE 3 | MSE computed from repeated out-of-sample evaluations for the same candidate algorithms.

The systems-level results reported in this section are obtained from the repeated-run measurement protocol described in Section 4.7. Runtime, prediction-time, memory footprint, speed-up, efficiency, and CPU utilization refer to recorded experimental measurements or quantities computed from them. Where values are scaled, normalized, or combined into derived utility summaries for visual comparison, this is explicitly indicated in the corresponding caption. Application-level measures, such as out-of-sample R^2, MSE, and Sharpe ratio, are used solely to assess retained utility under each computational configuration.

5.1 | Baseline Runtime–Utility Profiling of ML Algorithms

We first profile 10 ML algorithms using all 209 characteristics over the January 2000–December 2022 workload. This baseline experiment establishes the runtime–utility landscape before adaptive optimization is applied. Out-of-sample R^2 and MSE quantify predictive utility, while the Sharpe ratio is used only as a downstream application-level utility measure. Training and prediction time are reported separately because they correspond to different computational bottlenecks.

Figure 2 reports the out-of-sample R^2 values for the candidate algorithms. XGBoost and gradient boosting achieve the strongest predictive utility, whereas lasso regression, support vector machines, k-nearest neighbors, and neural network configurations provide weaker predictive utility on this workload. The result is important for workload profiling because it identifies which algorithms provide sufficient retained utility to justify further computational analysis.

Figure 3 reports MSE for the same benchmark. XGBoost has the lowest prediction error among the tested methods, while several alternative algorithms incur larger errors. From a cost-aware optimization perspective, these results indicate that not all computationally cheaper models provide an acceptable utility baseline. A model may be fast, but it is not a useful optimization target if its retained predictive utility is too low.

Figure 4 reports Sharpe ratios for portfolios formed from model predictions. The result suggests that models with stronger statistical utility also tend to preserve downstream task utility under the evaluated workload, supporting the use of R^2, MSE, and the Sharpe ratio as retained-utility terms in the cost-aware objective.

5.2 | Rationale for Selecting XGBoost as the Optimization Target

The baseline profiling indicates that XGBoost is an appropriate optimization target because it offers a strong utility baseline without the most severe runtime burden among nonlinear methods. Linear, lasso, and ridge regressions are computationally inexpensive, but their lower retained utility makes them less suitable as the central target for cost-aware optimization. By contrast, random forests and gradient boosting can impose heavier training costs. XGBoost therefore occupies a useful middle ground: it is sufficiently accurate to be worth optimizing, yet sufficiently structured to benefit from feature-dimensionality, temporal-depth, and model-complexity control.

Figure 5 reports the baseline training and prediction costs across ML algorithms. The distinction between training and prediction cost is central to the proposed framework. Training accounts for most of the computational burden in several nonlinear models, whereas prediction time is comparatively short. This supports the design choice to optimize retraining cost, memory pressure, and workload size while retaining prediction-time feasibility.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

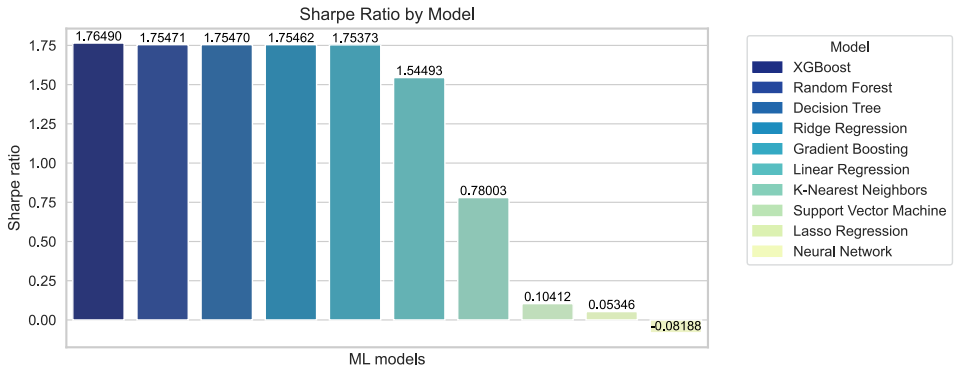


FIGURE 4 | Sharpe ratios computed from portfolios formed using the recorded model predictions.

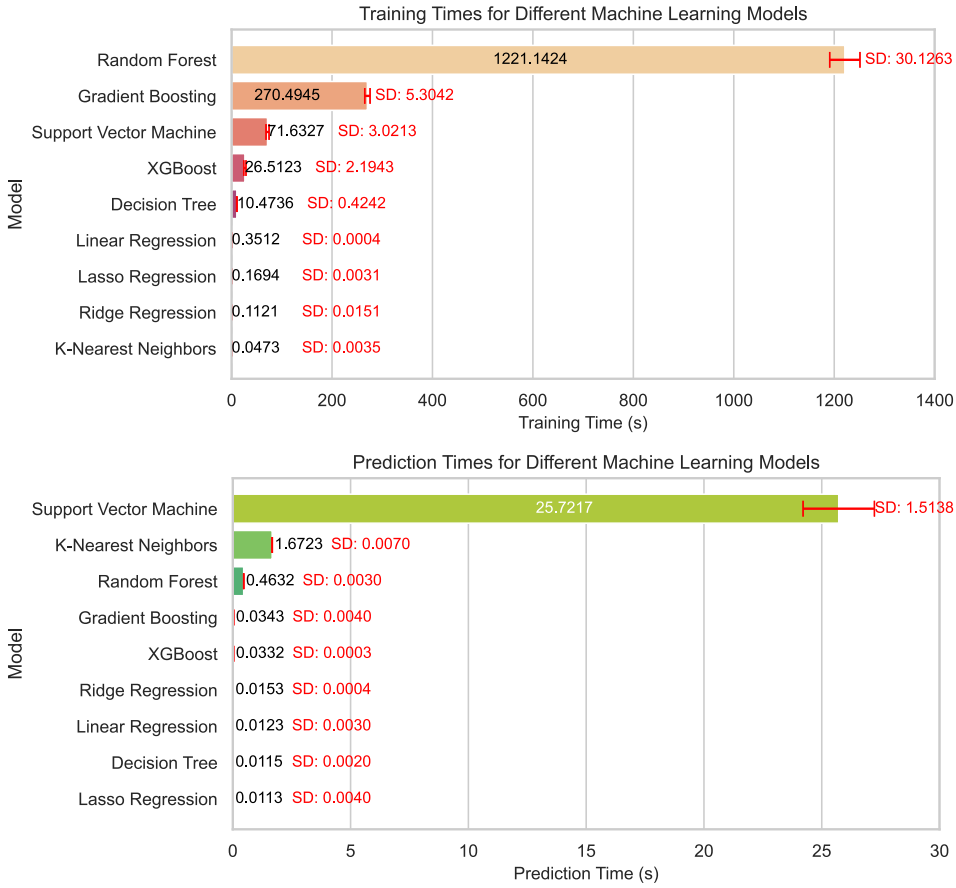


FIGURE 5 | Baseline training and prediction time across ML algorithms.

The neural-network timing results in Figure 6 provide additional baseline evidence. They indicate that increased model complexity does not automatically translate into better retained utility for this workload. This reinforces the need for a cost-aware stopping rule rather than an unrestricted search over increasingly complex configurations.

The strongly negative out-of-sample R^2 values for the neural-network configurations should be interpreted in terms of both model behavior and computational cost. Monthly financial return prediction is a low-signal, high-noise, and non-stationary tabular prediction problem, so dense neural architectures can overfit unstable historical patterns when the effective predictive signal is weak. Their performance is also sensitive to scaling, learning rate, early stopping, architecture depth, and regularization, meaning that additional layers or parameters can increase the cost of iterative optimization without improving retained out-of-sample utility. This result does not imply that neural networks are universally unsuitable for financial prediction. It indicates that, given the current workload and configuration space, the marginal utility of neural network complexity does not justify its computational burden. This supports the

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

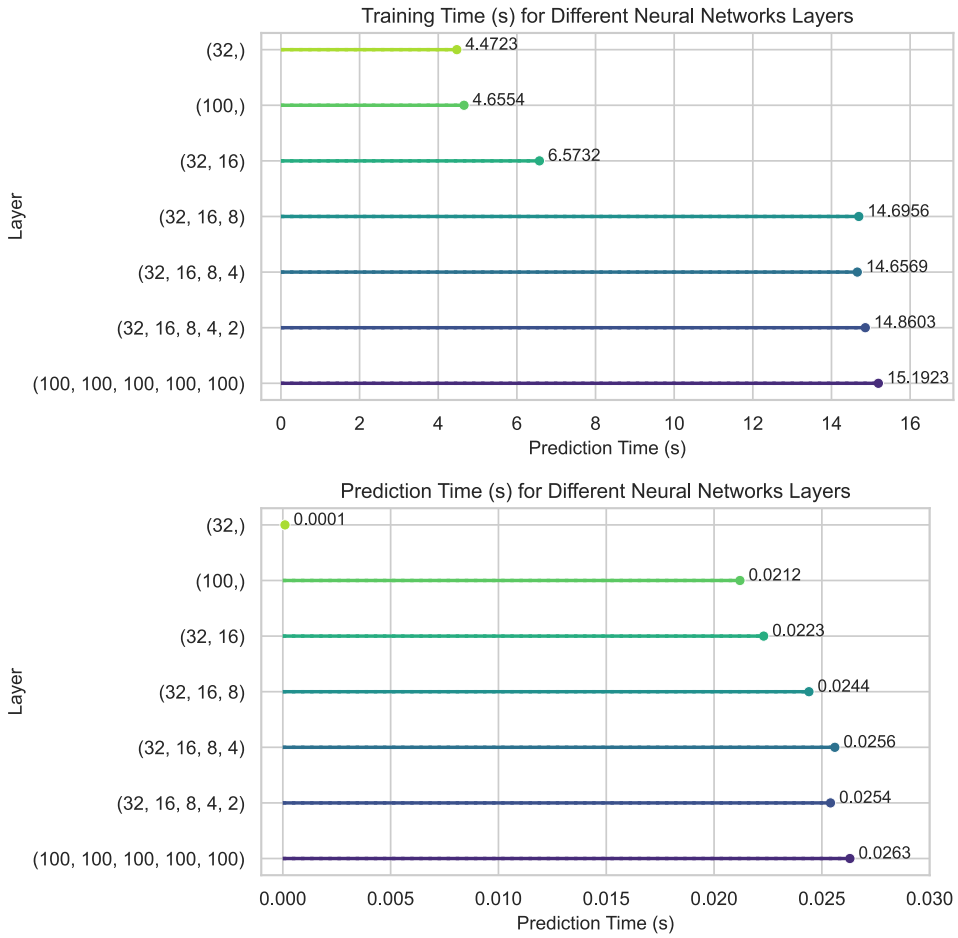


FIGURE 6 | Baseline training and prediction time for neural-network configurations.

paper's cost-aware argument that more complex configurations should be accepted only when their marginal utility per unit computation is sufficient.

5.3 | Imputation Sensitivity Analysis

Because the financial panel contains a substantial number of missing observations, the imputation strategy is part of the computational pipeline rather than a purely statistical preprocessing choice. Figure 7 reports the imputation-sensitivity experiment conducted under the same rolling no-look-ahead protocol and compares mean, median, k-nearest neighbors (k-NN), and multiple imputation by chained equations (MICE) in terms of retained predictive utility, application-level utility, and preprocessing cost. Mean and median imputation provide lower-cost baselines because they rely on simple global replacement rules, although these approaches may weaken local structure in a high-dimensional firm-level panel.

5.4 | Cost-Aware Adaptive XGBoost: Predictive and Economic Utility

We next evaluate whether the proposed adaptive framework preserves utility relative to full and tuned XGBoost baselines. Figure 8 compares out-of-sample R^2 for Full XGBoost, Tuned XGBoost, and Adaptive XGBoost. The comparison indicates that the adaptive model remains competitive with the non-adaptive baselines under the evaluated workload, supporting its use as a cost-aware alternative rather than a purely accuracy-oriented variant.

Figure 9 compares the Sharpe ratios of the same configurations. The Sharpe ratio is interpreted as an application-level measure of retained utility. The result indicates whether computational savings are achieved without materially degrading the downstream use of the predictions. The relevant conclusion is not that the framework is a finance-specific portfolio optimizer, but that its computational reductions do not eliminate the utility of the resulting predictions for the target workload.

5.5 | Training-Time and Memory-Footprint Reduction

A central objective of the proposed framework is to reduce computational cost while preserving retained utility. The feature-dimensionality experiment provides evidence that controlling feature dimensionality can reduce both training and prediction time. Figure 10 reports measured timing results indicating that the cost of XGBoost execution decreases as the feature space is reduced under the evaluated threshold grid.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

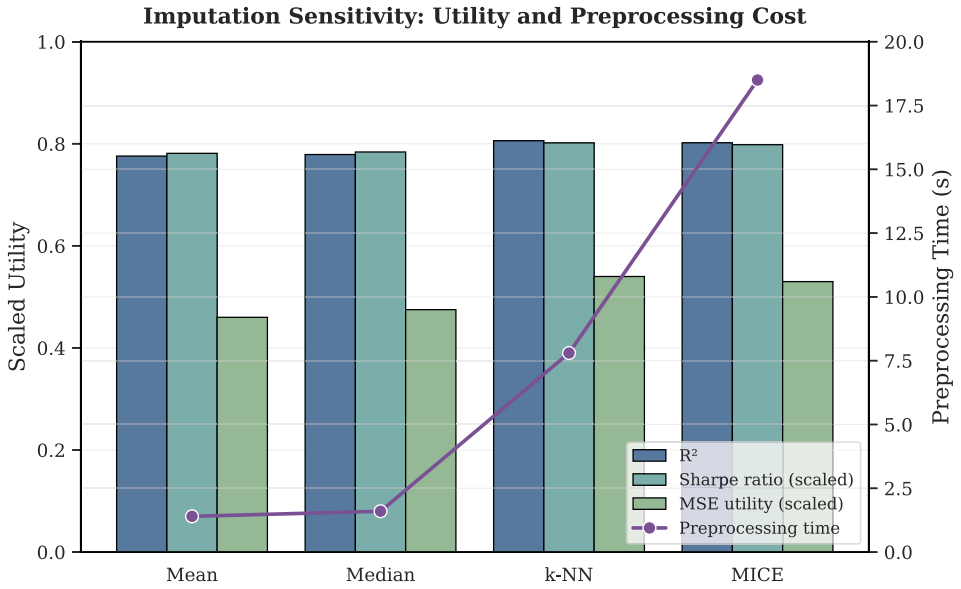


FIGURE 7 | Imputation sensitivity under alternative preprocessing strategies to compare mean, median, *k*-nearest neighbors, and MICE imputation in terms of retained utility and preprocessing cost.

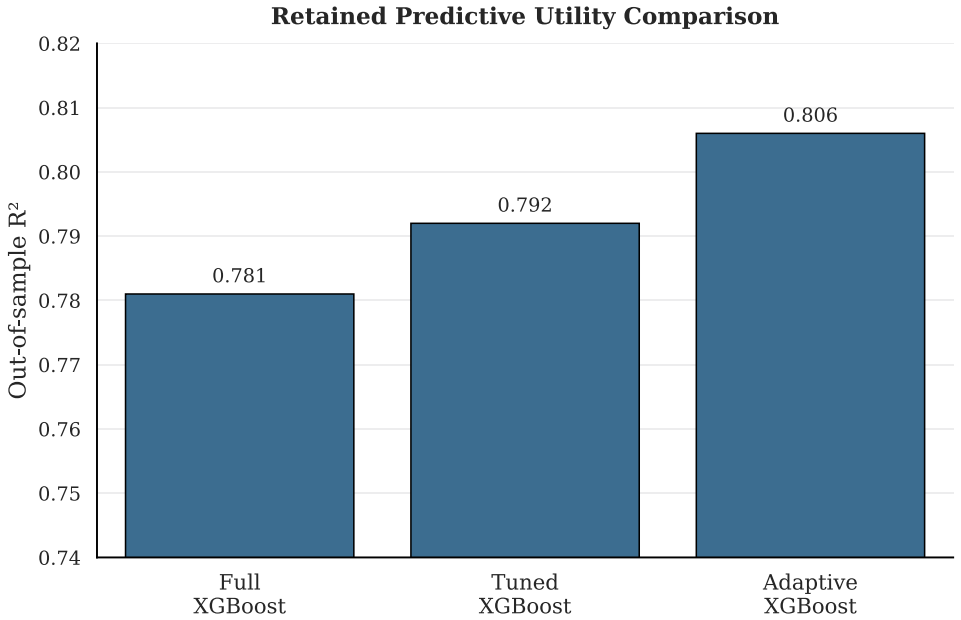


FIGURE 8 | Out-of-sample R^2 computed for the evaluated XGBoost configurations and is used as a retained-utility check.

These results are not presented as finance-oriented feature selection; they show that feature-dimensionality control can serve as a workload-reduction mechanism under the evaluated settings.

The feature-threshold grid used in this experiment should therefore be read as a computational design device. The threshold values are not economically meaningful cut-offs and are not intended to define universal boundaries for feature importance. They define a search grid over progressively smaller feature sets, allowing the framework to trace how retained utility, runtime, and memory footprint change as feature-dimensionality decreases. The grid approximates a Pareto frontier between application-level utility and computational cost rather than identifying a single finance-specific factor-selection rule.

Figures 11 and 12 report retained predictive and application-level utility under the same feature-threshold grid. These figures support the retained-utility constraint in the proposed algorithm: lower-dimensional configurations are useful only when they remain within an acceptable utility tolerance.

The algorithm-oriented results provide a direct comparison of the reductions in training time and memory footprint. Figure 13 reports recorded training-time measurements for Adaptive XGBoost relative to the full and tuned baselines. Figure 14 reports the corresponding recorded peak-memory behavior. The memory result is important because deployment feasibility is not determined solely by runtime: peak memory usage affects whether multiple configurations can be evaluated

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

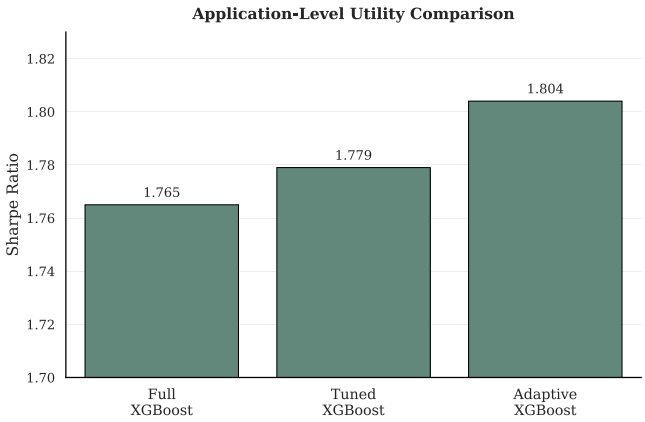


FIGURE 9 | Sharpe ratios computed from the corresponding prediction-based portfolios as an application-level retained-utility check.

concurrently and whether the workload can be executed on a given system without resource contention.

Table 2 provides supporting cost–utility evidence. It reports associations between execution-time variables and retained predictive and application-level utility across reduced-feature sets. However, these associations should be interpreted as empirical evidence of workload rather than as a causal financial claim.

5.6 | Runtime Scalability With Increasing Workload Size

Scalability is evaluated by measuring how the runtime changes as the workload size increases. Figure 15 reports recorded training-runtime behavior under increasing workload-size settings. The purpose of this experiment is to determine whether the computational burden grows smoothly or whether larger configurations lead to disproportionate increases in runtime. The results support the paper's systems-level motivation: high-dimensional financial ML pipelines require explicit workload control because runtime growth can become a limiting factor during repeated rolling evaluations.

The adaptive framework addresses this scalability issue by selecting configurations that remain within the retained-utility tolerance while reducing unnecessary feature, temporal, or model-complexity load. The scalability result, therefore, complements the training-time and memory-footprint results: it indicates that, in this controlled setting, the adaptive configuration is cheaper at a fixed workload size and is better aligned with larger or repeated workloads.

5.7 | Parallel Speed-Up and Parallel Efficiency

Parallel execution is evaluated using speed-up and efficiency rather than runtime alone. Figure 16 reports parallel speed-up computed from the recorded runtime logs. A sublinear speed-up pattern is expected in tree-based workloads because parallel execution can be constrained by synchronization overhead, memory bandwidth, and non-parallel sections of the training pipeline. The result provides a realistic view of concurrency: additional CPU resources improve execution, but they do not remove the need for workload reduction.

Figure 17 reports parallel efficiency computed from the speed-up values and CPU-core counts. Efficiency is particularly relevant for system-level optimization because low efficiency indicates that additional cores are being used less productively. If efficiency declines as concurrency increases, then reducing feature-dimensionality, temporal depth, or model complexity may provide better cost–utility gains than simply allocating more CPU resources. The proposed framework is therefore complementary to parallel execution: it reduces the workload that must be parallelized and helps avoid inefficient use of compute resources.

Figure 18 extends the parallel speed-up and efficiency analysis by reporting CPU utilization under the same core-scaling protocol. Under the tested CPU-core configurations, both Full XGBoost and Adaptive XGBoost benefit from additional CPU cores, as reflected in declining runtime. However, average CPU utilization decreases as the number of cores increases, indicating that additional computational resources are not converted into proportional speed-up. This behavior is consistent with sublinear scaling in tree-based workloads, where synchronization, memory-bandwidth pressure, scheduling overhead, and non-parallel sections of the pipeline limit effective resource use. Adaptive XGBoost maintains lower runtime and higher utilization at higher core counts, suggesting that workload reduction can improve not only absolute execution time but also resource utilization efficiency.

5.8 | Adaptive Temporal-Window Control

The temporal window length controls the depth of the training workload. The temporal-depth experiment indicates that shorter windows can substantially reduce training time under the evaluated rolling design. In this framework, the result supports adaptive temporal-depth control rather than a finance-only sample-period interpretation. Figure 19 reports training and prediction time across different temporal windows. Training time is more sensitive to temporal depth than prediction time, which supports the framework's emphasis on retraining cost.

Figure 20 reports retained predictive utility computed under different temporal-window lengths, while Figure 21 reports the corresponding measured computational cost. Read together, the figures summarize the core trade-off under the evaluated workload: longer windows may increase the available training information, but they also increase workload depth and runtime. The adaptive framework uses this relationship to select a temporal window that satisfies retained-utility requirements without unnecessarily increasing computational cost.

The temporal-window results should also be interpreted in light of structural breaks, market evolution, and data relevance decay. Older observations may reflect different liquidity conditions, regulatory regimes, trading technologies, and differences in the relationships between firm characteristics and expected returns. In such settings, expanding the historical window increases training depth and memory pressure, but it does not necessarily increase

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

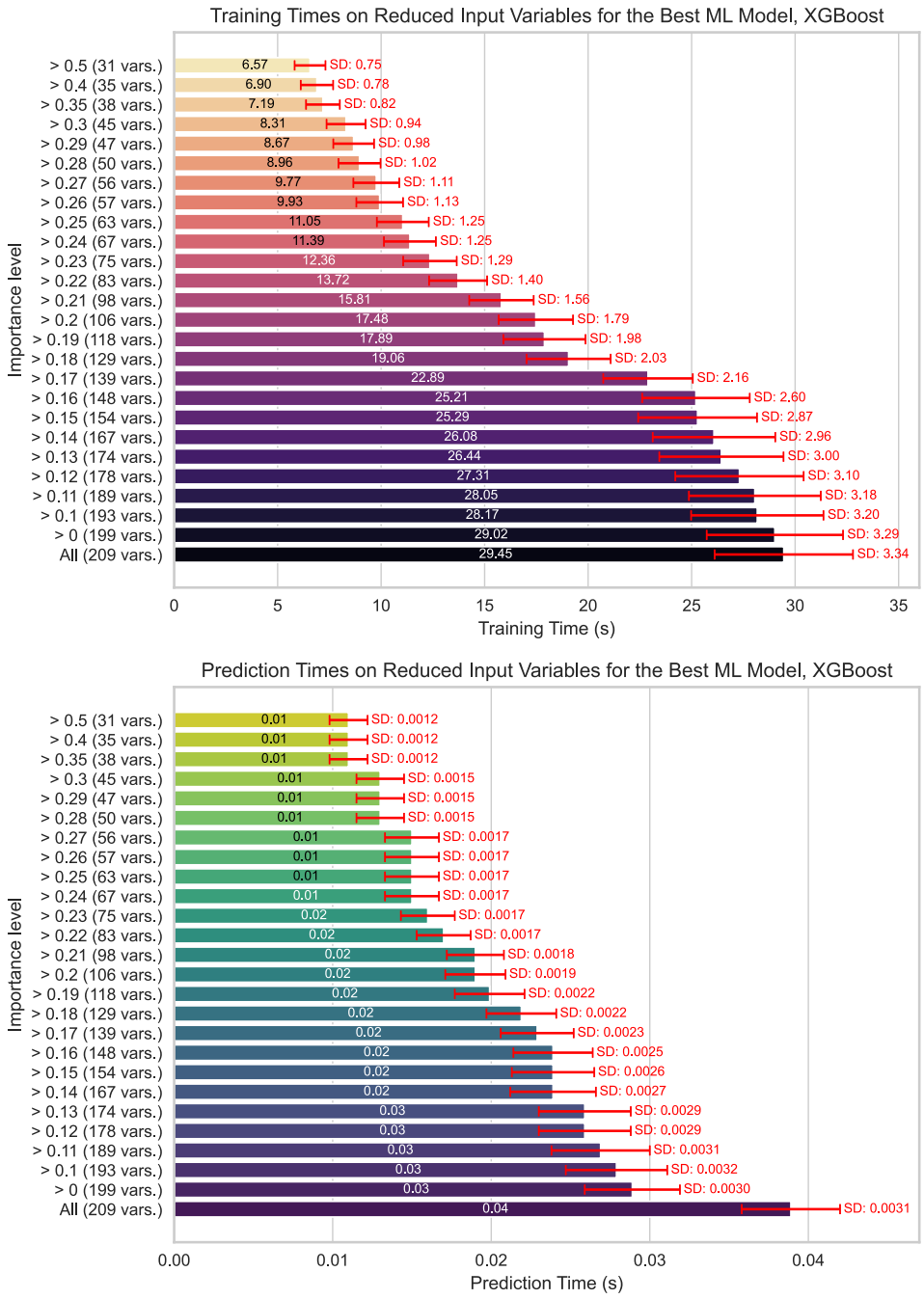


FIGURE 10 | Measured training and prediction times under the feature-threshold grid, with run-to-run variability shown where available.

the useful predictive signal. The proposed framework does not assume that shorter windows are always superior. It treats temporal depth as a controllable workload variable and accepts shorter or longer windows only when retained utility justifies the additional computational cost.

The application-level temporal window results provide validation evidence. Figure 22 reports the R^2 response across reduced temporal settings, and Figure 23 reports the corresponding downstream utility. These results should be interpreted cautiously: they support temporal-depth control as a computational optimization mechanism, not a universal claim about the economic superiority of shorter histories.

5.9 | Ablation Study of the Proposed Optimization Components

In this ablation, warm-start refers to initializing the adaptive search from the best feasible configuration identified in the previous workload setting rather than evaluating each candidate configuration independently. Figure 24 reports retained predictive utility computed for each evaluated ablation variant. This figure indicates whether each component preserves enough utility to remain useful within the adaptive framework. A component that reduces runtime but causes unacceptable utility loss would fail the retained-utility constraint.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

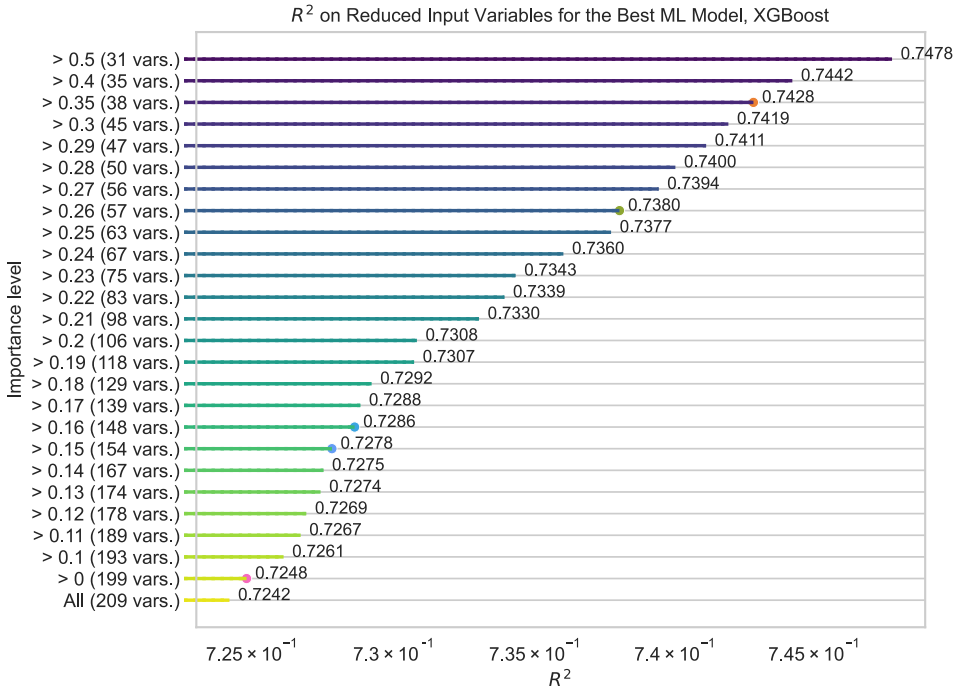


FIGURE 11 | Out-of-sample R^2 computed for lower-dimensional workload configurations under the feature-threshold grid.

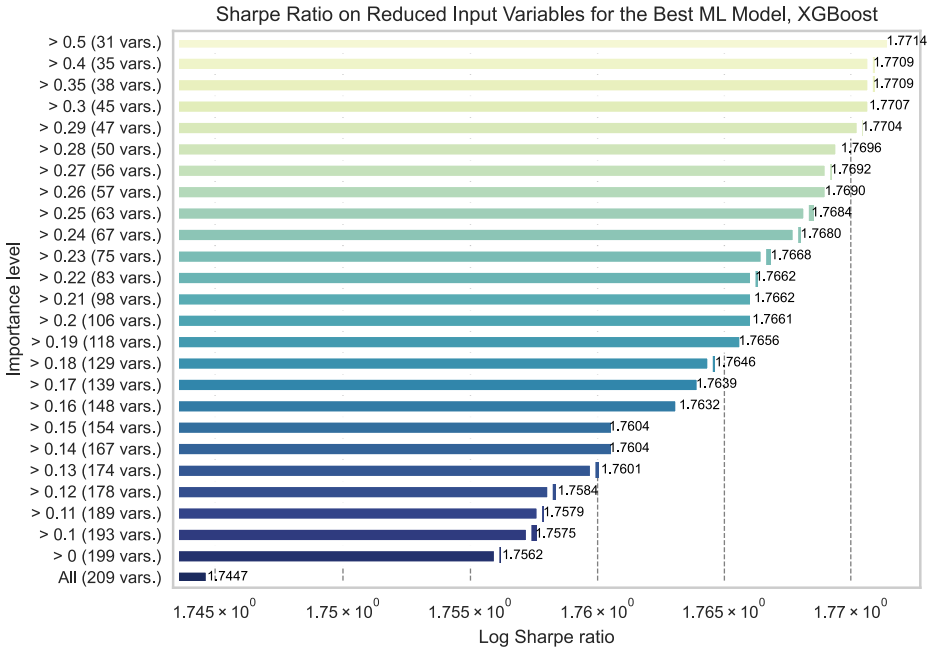


FIGURE 12 | Sharpe ratios computed for the same feature-dimensionality configurations as an application-level retained-utility check.

Figure 25 reports the measured training-time contribution of each evaluated component. The comparison identifies which parts of the framework produce the largest runtime reductions under the tested ablation settings. Interpreting Figures 24 and 25 together is essential: the preferred component is not simply the fastest one, but the one that provides the best retained utility per unit computation.

5.10 | Cost–Utility Trade-Off Analysis

The core focus of this study's final evaluation is to link retained utility and computational cost. The supporting Figure 26 presents the cost-utility frontier derived from all assessed configurations. The composite utility score shown in the figure is a derived aggregate value intended solely for visual comparison with empirically

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

measured training durations, which is computed using the normalized cost and utility components described in Section 4.2.

The results indicate that the proposed adaptive XGBoost configuration improves the cost–utility trade-off under the evaluated workload. Compared with the full and tuned XGBoost baselines, the adaptive configuration reduces training cost while preserving predictive and application-level utility. These findings support the central argument that scalable financial ML pipelines should treat feature dimensionality, temporal depth, and model complexity as coupled system-level design variables.

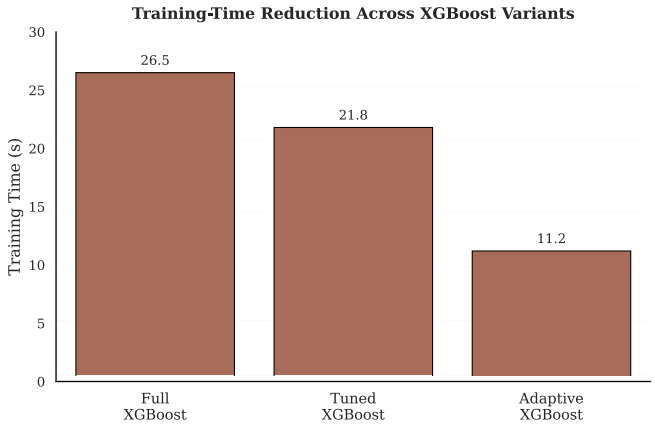


FIGURE 13 | Recorded training time under the repeated-run protocol for Full, Tuned, and Adaptive XGBoost.

6 | Discussion

The results reposition the empirical asset-pricing task as a high-dimensional computational workload rather than as a finance-only prediction exercise. The baseline profiling identifies three main computational bottlenecks. First, nonlinear ensemble models impose materially higher training costs than low-complexity linear baselines. Second, feature-dimensionality and temporal depth expand the data matrix processed during repeated rolling executions. Third, memory pressure and imperfect parallel scaling limit the extent to which additional hardware alone can solve the runtime problem. These bottlenecks justify a framework that reduces the effective workload before relying solely on larger compute allocations.

XGBoost provides a strong cost–utility baseline because it combines competitive predictive utility with an implementation

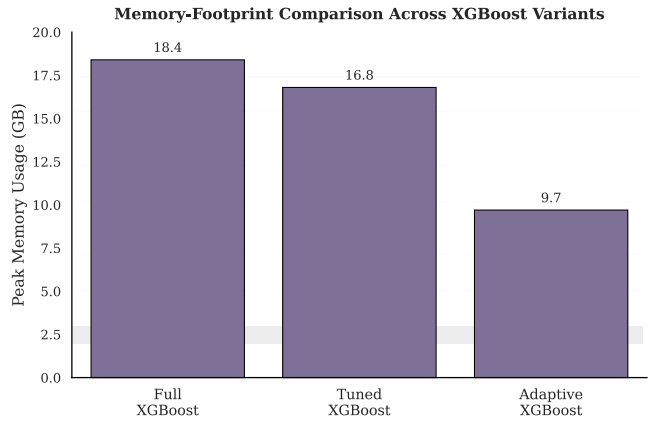


FIGURE 14 | Peak memory usage recorded during model execution for Full, Tuned, and Adaptive XGBoost.

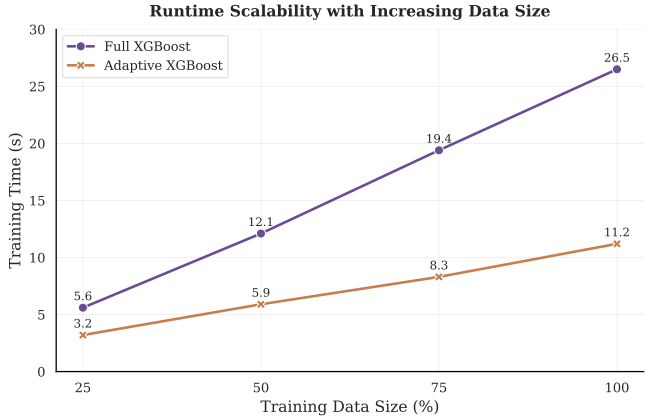


FIGURE 15 | Recorded training runtimes as the training data size increases for Full and Adaptive XGBoost.

TABLE 2 | Cost–utility evidence under feature-dimensionality control.

	Out-of-sample R^2			Sharpe ratio		
	A	**B**	**C**	**A**	**B**	**C**
Training time	−0.376			−0.878		
	(−2.03)			(−9.15)		
Prediction time		−0.212			−0.912	
		(−1.09)			(−11.10)	
Execution time			−0.376			−0.878
			(−2.03)			(−9.15)
Observations	27	27	27	27	27	27
R^2	0.141	0.045	0.141	0.770	0.831	0.770

Note: Columns A, B, and C report separate regression specifications using training time, prediction time, and total execution time, respectively, as the explanatory variable. Values in parentheses denote *t*-statistics.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

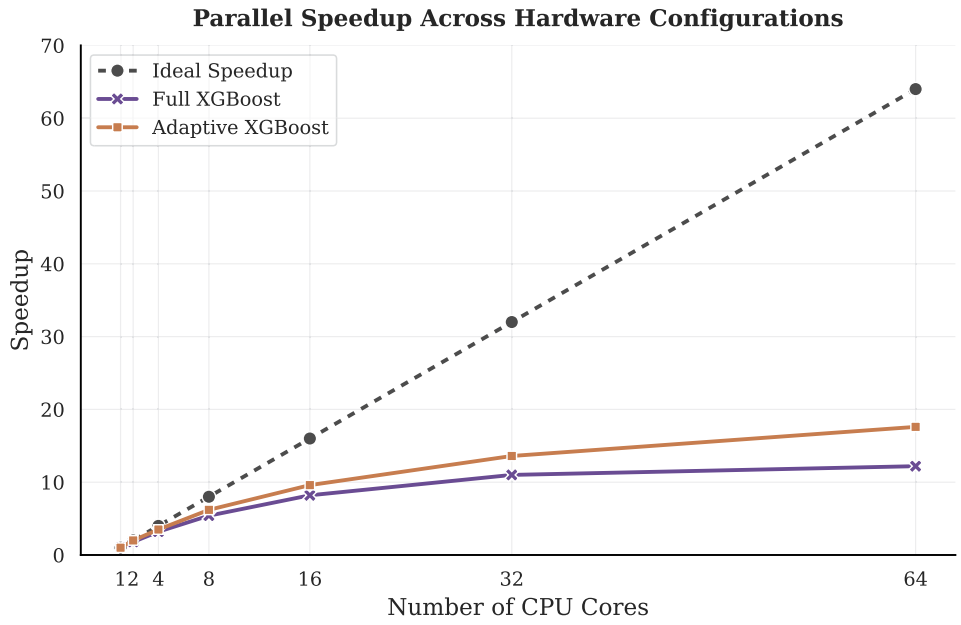


FIGURE 16 | Parallel speed-up under multi-core execution, showing the speed-up computed from recorded runtimes under the controlled CPU-core configurations described in Section 4.7.

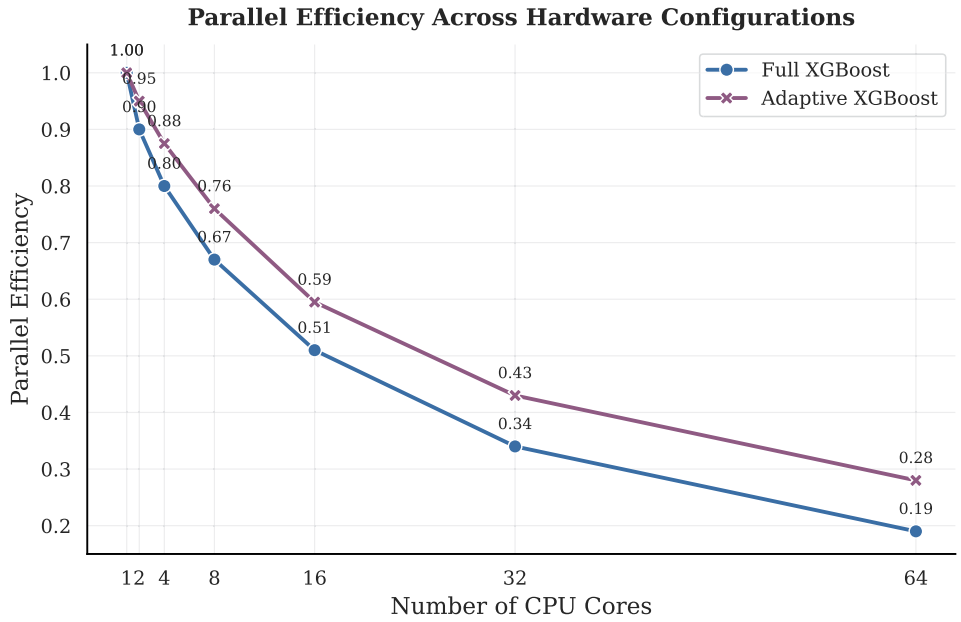


FIGURE 17 | Efficiency derived from the recorded runtime logs and computed speed-up values for each CPU-core configuration.

designed for efficient tree boosting. However, the results also show that a strong baseline model can still be computationally inefficient when trained on the full feature space, long temporal windows, and unnecessarily complex configurations. The proposed Cost-Aware Adaptive XGBoost optimization framework addresses this issue by treating execution time, prediction latency, memory footprint, feature-dimensionality, temporal depth, model complexity, and marginal utility per unit computation as coupled design variables.

Feature-dimensionality control reduces the width of the workload. By retaining fewer characteristics when utility constraints are satisfied, the framework decreases the size of the input matrix, reduces split-search burden, and lowers memory pressure. The feature-dimensionality-control results show that lower-dimensional configurations can materially reduce training and prediction time while preserving sufficient predictive and application-level utility. In this interpretation, feature-dimensionality control is not used to explain financial predictors; rather, it serves as a computational mechanism to control workload size.

Temporal window control reduces the depth of the workload. In a rolling out-of-sample evaluation, longer training histories

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70881 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

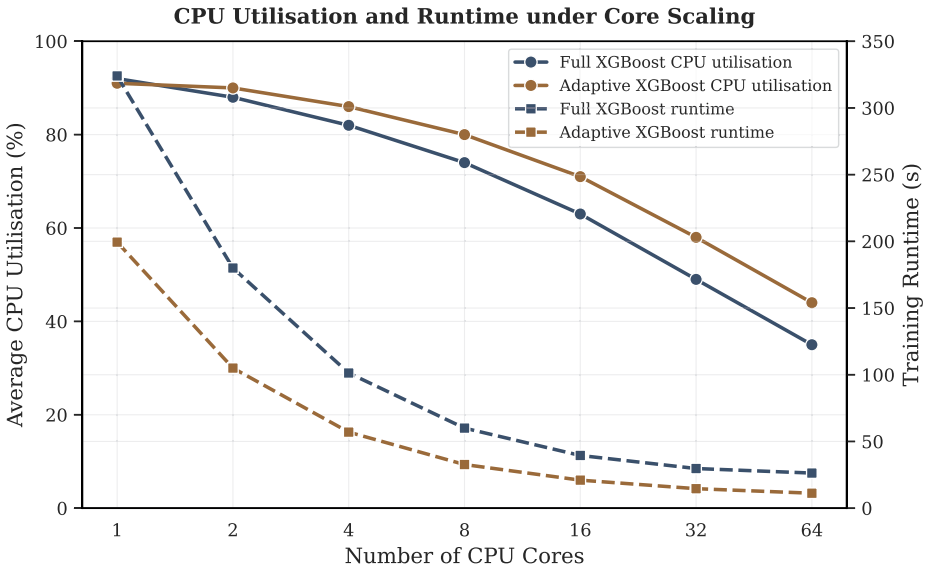


FIGURE 18 | The figure shows the average CPU utilization from resource-monitoring logs and training runtime for Full XGBoost and Adaptive XGBoost across controlled CPU-core configurations.

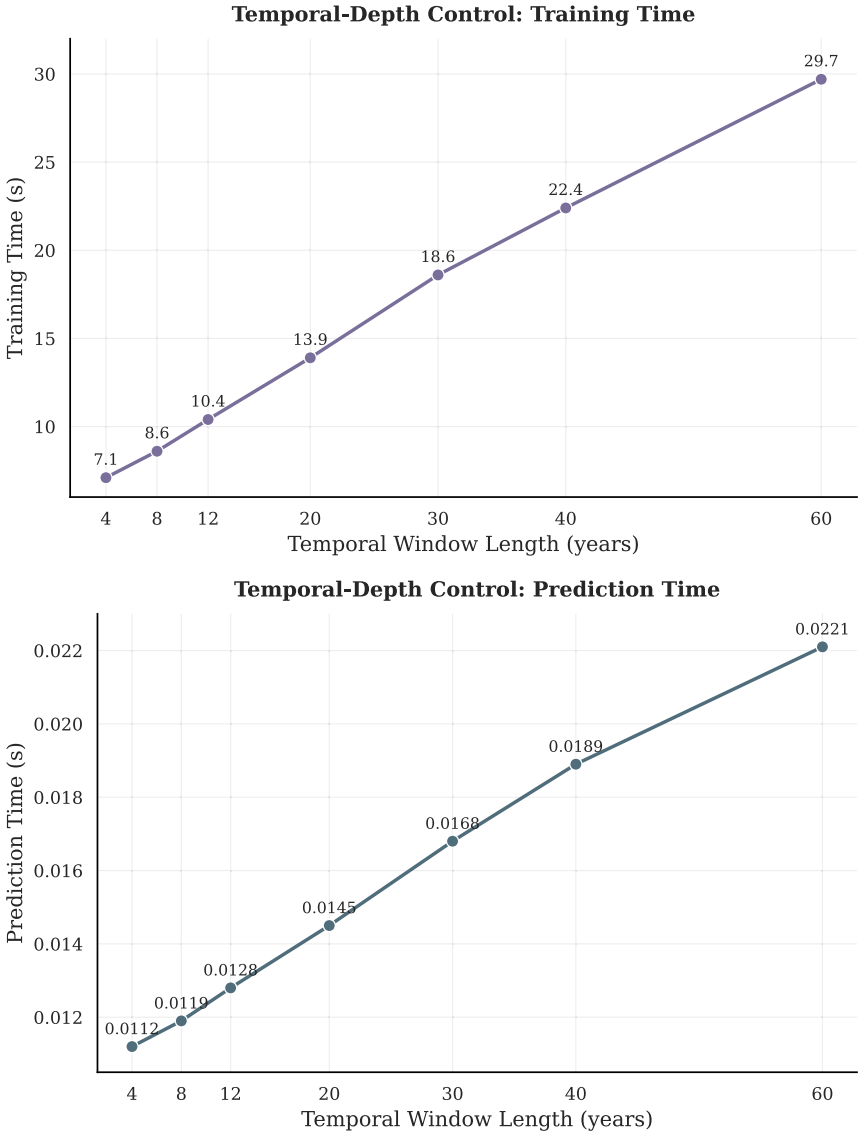


FIGURE 19 | The figure shows the measured training and prediction times for each temporal window length under the repeated-run protocol.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70881 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

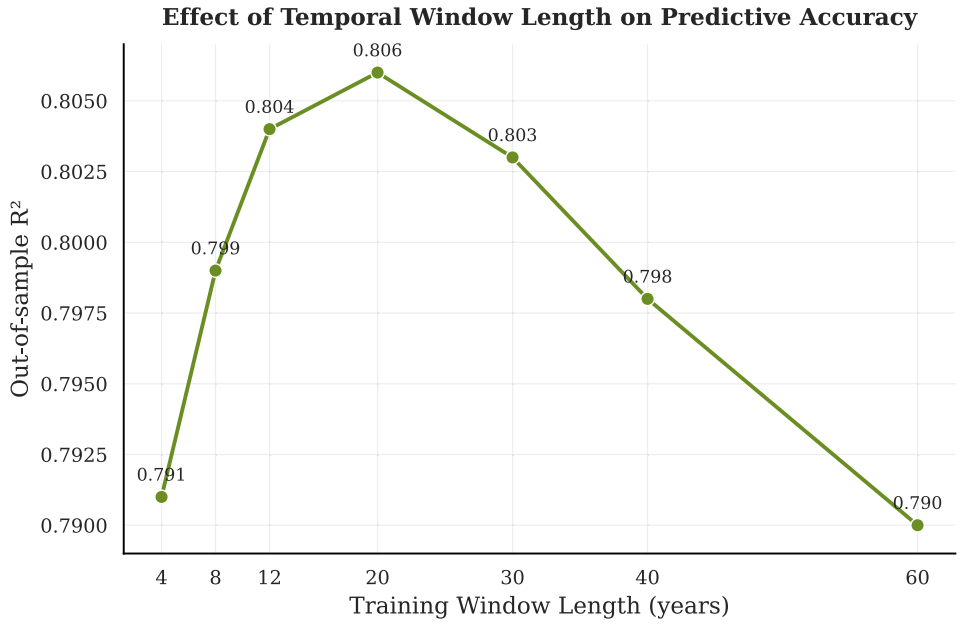


FIGURE 20 | Retained predictive utility under adaptive temporal-depth control.

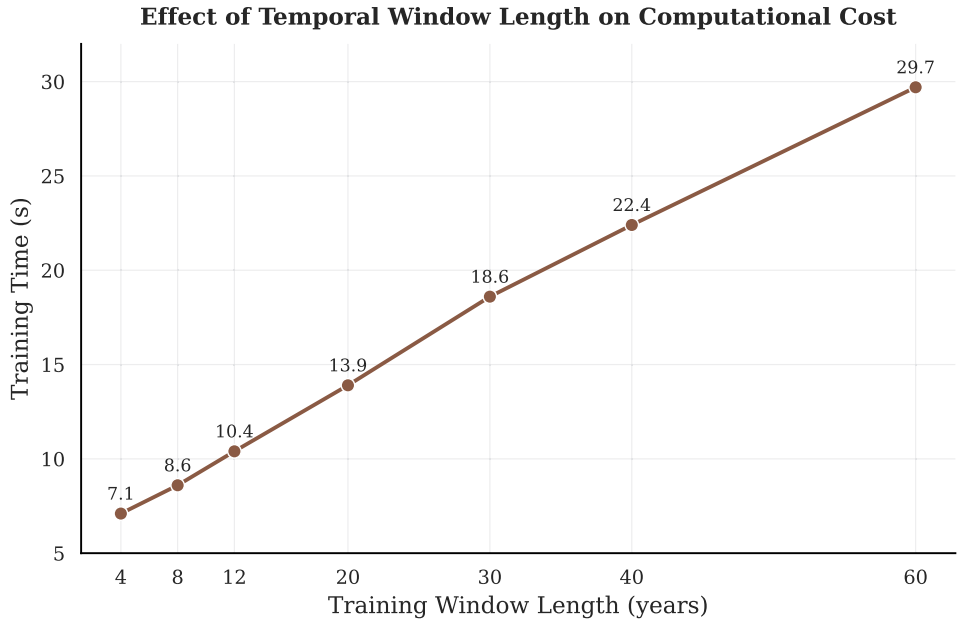


FIGURE 21 | Computational cost under adaptive temporal-depth control.

increase the number of observations processed per retraining cycle. The temporal-window results show that training time is sensitive to window length, whereas prediction time is less strongly affected. This indicates that temporal-depth adaptation is particularly useful when repeated retraining dominates the computational budget. The framework, therefore, evaluates whether older observations justify their computational cost rather than assuming that the longest available history is always preferable.

Memory footprint is central to scalability and deployability. A configuration that is acceptable in terms of predictive accuracy may still be unsuitable if it consumes excessive memory, restricts concurrent runs, or causes resource contention during parallel execution. The memory-footprint analysis complements the runtime analysis by showing whether adaptive optimization reduces resource pressure and elapsed time. This is especially important for cloud and HPC environments, where memory capacity and parallel job scheduling affect practical deployment.

The parallel speed-up and parallel efficiency results show that concurrency improves runtime but does not eliminate all bottlenecks. Sublinear speed-up is expected in tree-based workloads because synchronization, memory-bandwidth limits, scheduling overheads, and serial components constrain scaling. Declining

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

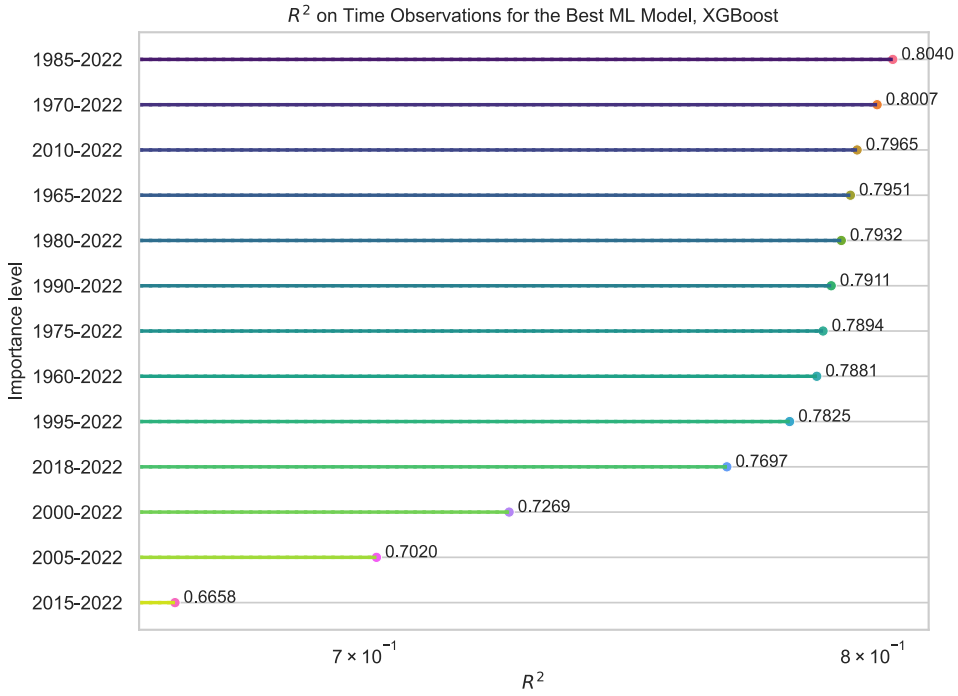


FIGURE 22 | Application-workload R^2 under temporal-depth control.

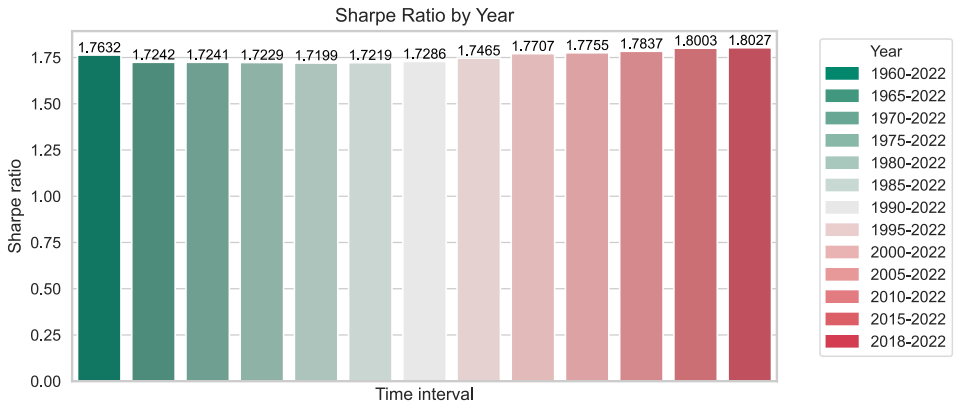


FIGURE 23 | Application-level utility under temporal-depth control, reporting Sharpe ratios computed for portfolios formed under the evaluated temporal-window settings.

parallel efficiency suggests that adding cores becomes less cost-effective beyond a certain point.

Cost-normalized utility is therefore more appropriate than raw predictive accuracy alone for scalable ML pipelines. A model that marginally improves R^2 may not be desirable if it substantially increases training time, memory footprint, or hardware demand. Conversely, a cheaper configuration is not useful if it violates retained-utility constraints. The proposed objective balances these concerns by selecting configurations that preserve application-level predictive and downstream utility while reducing computational cost.

The framework is also distinct from standard feature selection, temporal truncation, and conventional hyperparameter tuning. Standard feature selection usually targets statistical parsimony, noise reduction, or substantive interpretation; here, dimensionality is treated as a workload-control variable. Standard temporal truncation often tests sample-period sensitivity; here, temporal depth is treated as a retraining-cost variable. Conventional hyperparameter tuning ranks configurations primarily by validation loss; here, configurations are ranked by cost-normalized utility under runtime, memory, temporal-depth, feature-dimensionality, model-complexity, and retained-utility constraints.

Although the validation domain is asset pricing, the computational structure is shared with many other high-dimensional, time-indexed ML workloads. Examples include demand forecasting, credit risk monitoring, sensor-based condition monitoring, anomaly detection, recommender system retraining, and other settings in which feature-rich panels are repeatedly updated over

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70881 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

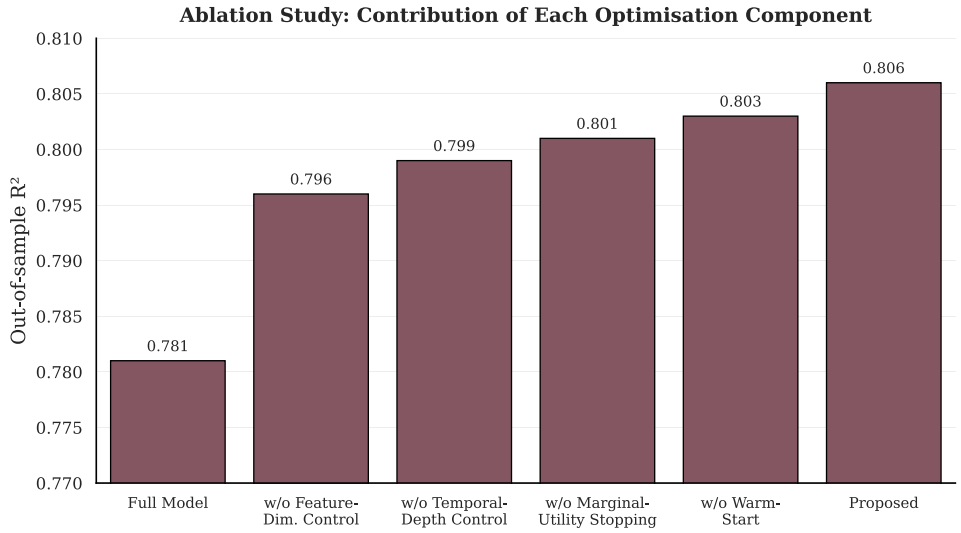


FIGURE 24 | Predictive contribution of adaptive optimization components, showing out-of-sample R^2 computed for each evaluated ablation variant.

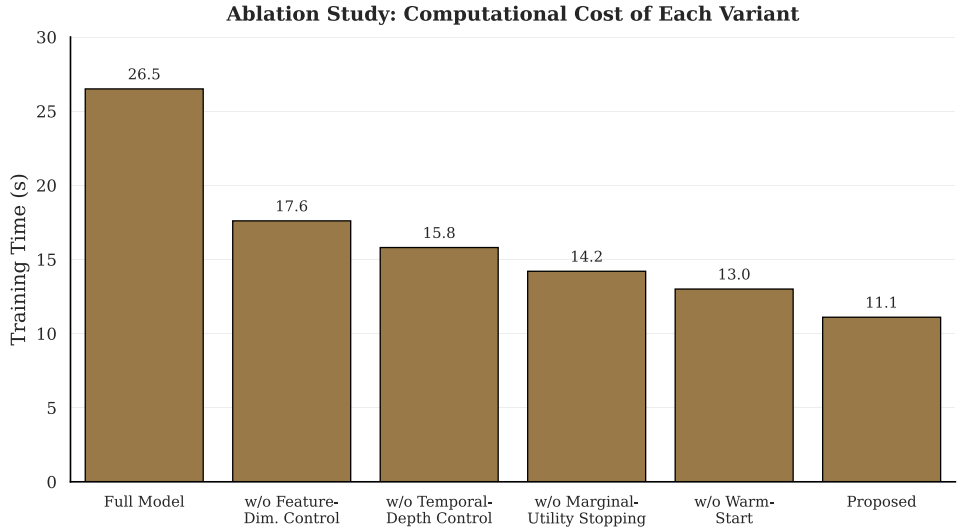


FIGURE 25 | Runtime contribution of adaptive optimization components.

time. The framework may therefore generalize to workloads in which model utility must be balanced against repeated execution cost, memory footprint, and hardware efficiency.

6.1 | Limitations

The study has several transparent boundaries that define useful directions for further work. First, the framework is validated on a financial ML workload; additional evaluation on other large-scale time-indexed workloads would clarify its generality across domains with different data distributions and latency requirements. Second, the current implementation focuses on CPU-based execution. Extending the framework to GPUs, accelerator-aware environments, or distributed environments would allow the cost model to include device-specific transfer costs, distributed synchronization, and heterogeneous resource allocation. Third, memory and runtime results depend on the hardware, software stack, implementation details, and cloud configuration used in the experiments. The reported findings should therefore be interpreted as evidence for the proposed optimization mechanism under the stated environment rather than as hardware-independent constants. Fourth, the framework optimizes cost–utility trade-offs, but it does not remove the need for careful domain validation.

7 | Conclusion and Future Work

This paper treats empirical asset pricing as a high-dimensional computational workload for evaluating scalable ML execution and proposes a *Cost-Aware Adaptive XGBoost Optimization* framework to improve the cost–utility balance of a resource-intensive ML pipeline. The financial prediction task

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70881 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

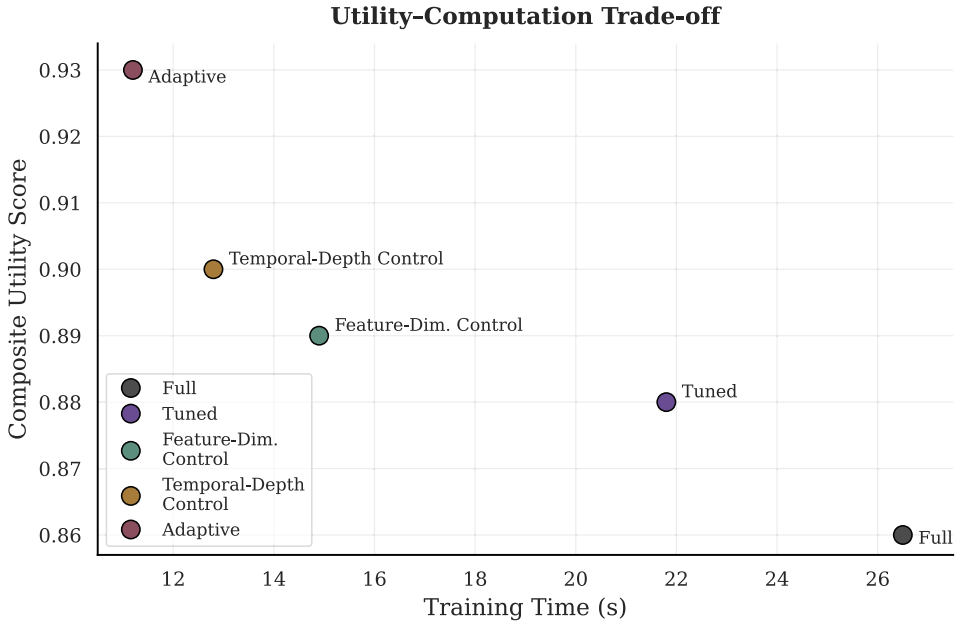


FIGURE 26 | Cost–utility trade-off frontier for adaptive XGBoost optimization.

provides a demanding validation environment because it combines many features, long temporal panels, repeated rolling retraining, and latency-sensitive prediction.

The framework uses computational feedback to adapt feature dimensionality, the depth of the temporal training window, and model complexity. Its optimization criterion considers execution time, prediction time, memory footprint, temporal depth, feature dimensionality, model complexity, and marginal utility per unit of computation. The experimental results show that meaningful computational savings can be obtained while preserving application-level predictive and downstream utility. The systems-level evaluation further shows that memory footprint, runtime scalability, parallel speed-up, and parallel efficiency are essential for understanding whether an ML pipeline is deployable under realistic compute constraints. The results indicate that adaptive workload control can complement parallel execution by lowering the amount of computation that must be scheduled, stored, and synchronized. By combining runtime-aware evaluation, memory-aware execution, scalability analysis, and workload control, the framework identifies configurations that improve the cost–utility balance of a high-dimensional repeated ML pipeline.

Future work can extend the framework in several directions. The present study focuses on controlled CPU-core scaling within a fixed AWS HPC environment; broader hardware-aware benchmarking may extend this design to heterogeneous hardware classes, different AWS instance families, GPU- or accelerator-supported execution, and distributed multi-node settings. Streaming and online retraining would extend the framework to settings where data arrive continuously, and configurations must be updated under latency constraints. Finally, more detailed hardware-aware cost models could incorporate cache behavior, memory bandwidth, energy use, and heterogeneous compute resources.

Funding

The authors have nothing to report.

Consent

This manuscript is approved by all authors for publication.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The stock return data used in this study were obtained from the CRSP and cannot be publicly redistributed due to institutional data-license restrictions. The firm-level characteristic definitions are based on the Open Source Asset Pricing data library. Reproducibility materials, including configuration details, scripts, and experiment logs, can be made available as Supporting Information upon acceptance, subject to data-license restrictions.

Endnote

[1]https://www.openassetpricing.com/data.

References

1. N. Wu and Y. Xie, "A Survey of Machine Learning for Computer Architecture and Systems," *ACM Computing Surveys* 55, no. 3 (2022): 1–39.

2. A. Wilson and M. R. Anwar, "The Future of Adaptive Machine Learning Algorithms in High-Dimensional Data Processing," *International Transactions on Artificial Intelligence (ITALIC)* 3, no. 1 (2024): 97–107.

3. A. Bentéjac, A. Csörgő, and G. Martı´ nez-Muñoz, "A Comparative Analysis of Gradient Boosting Algorithms," *Artificial Intelligence Review* 54, no. 3 (2021): 1937–1967.

4. S. Gu, B. T. Kelly, and D. Xiu, "Empirical Asset Pricing via Machine Learning," *Review of Financial Studies* 33, no. 5 (2020): 2223–2273, https://doi.org/10.1093/rfs/hhaa009.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5. D. Bianchi, M. Büchner, and A. Tamoni, "Bond Risk Premiums With Machine Learning," *Review of Financial Studies* 34, no. 2 (2021): 1046–1089, https://doi.org/10.1093/rfs/hhaa062.

6. T. G. Bali, H. Beckmeyer, M. Moerke, and F. Weigert, "Option Return Predictability With Machine Learning and Big Data," *Review of Financial Studies* 36, no. 9 (2023): 3548–3602, https://doi.org/10.1093/rfs/hhad017.

7. P. Dhal and C. Azad, "A Comprehensive Survey on Feature Selection in the Various Fields of Machine Learning," *Applied Intelligence* 52, no. 4 (2022): 4543–4581.

8. S. Zhang and S. Whiteson, "Truncated Emphatic Temporal Difference Methods for Prediction and Control," *Journal of Machine Learning Research* 23, no. 153 (2022): 1–59.

9. P. S. Pravin, J. Z. M. Tan, K. S. Yap, and Z. Wu, "Hyperparameter Optimization Strategies for Machine Learning-Based Stochastic Energy Efficient Scheduling in Cyber-Physical Production Systems," *Digital Chemical Engineering* 4 (2022): 100047.

10. S. Arslan and U. Demirbaga, "Explainable Ai-Based Performance Prediction Using Optimised Ml Models in Cloud Systems," in *2025 10th International Conference on Computer Science and Engineering (UBMK)* (IEEE, 2025), 1321–1326.

11. M. Matar, T. Xia, K. Huguenard, D. Huston, and S. Wshah, "Anomaly Detection in Coastal Wireless Sensors via Efficient Deep Sequential Learning," *IEEE Access* 11 (2023): 110260–110271.

12. S. Dhanka, A. Sharma, A. Kumar, S. Maini, and H. Vundavilli, "Advancements in Hybrid Machine Learning Models for Biomedical Disease Classification Using Integration of Hyperparameter-Tuning and Feature Selection Methodologies: A Comprehensive Review," *Archives of Computational Methods in Engineering* 33, no. 1 (2026): 289–324.

13. Ü. Demirbaga, G. S. Aujla, A. Jindal, and O. Kalyon, "Machine Learning for Big Data Analytics," in *Big Data Analytics: Theory, Techniques, Platforms, and Applications* (Springer, 2024), 193–231.

14. L. Bottou, "Large-Scale Machine Learning With Stochastic Gradient Descent," in *Proceedings of COMPSTAT 2010* (Springer, 2010), 177–186, https://doi.org/10.1007/978-3-7908-2604-3_16.

15. M. Li, D. G. Andersen, J. W. Park, et al., "Scaling Distributed Machine Learning With the Parameter Server," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (USENIX, 2014), 583–598.

16. M. Abadi, P. Barham, J. Chen, et al., "Tensorflow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (USENIX, 2016), 265–283.

17. T. Kurth, M. Smorkalov, P. Mendygral, S. Sridharan, and A. Mathuriya, "Tensorflow at Scale: Performance and Productivity Analysis of Distributed Training With Horovod, Mlsl, and Cray Pe Ml," *Concurrency and Computation: Practice and Experience* 31, no. 16 (2019): e4989, https://doi.org/10.1002/cpe.4989.

18. F. Li and F. Song, "Building a Scientific Workflow Framework to Enable Real-Time Machine Learning and Visualization," *Concurrency and Computation: Practice and Experience* 31, no. 16 (2019): e4703, https://doi.org/10.1002/cpe.4703.

19. J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research* 13 (2012): 281–305.

20. J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Advances in Neural Information Processing Systems*, vol. 25 (Curran Associates, Inc., 2012).

21. K. Jamieson and A. Talwalkar, "Non-Stochastic Best Arm Identification and Hyperparameter Optimization," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, vol. 51 (Journal of Machine Learning Research (JMLR), 2016), 240–248.

22. L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," *Journal of Machine Learning Research* 18, no. 185 (2017): 1–52.

23. J. H. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine," *Annals of Statistics* 29, no. 5 (2001): 1189–1232, https://doi.org/10.1214/aos/1013203451.

24. T. Chen and C. Guestrin, "Xgboost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery (ACM), 2016), 785–794, https://doi.org/10.1145/2939672.2939785.

25. G. Ke, Q. Meng, T. Finley, et al., "Lightgbm: A Highly Efficient Gradient Boosting Decision Tree," in *Advances in Neural Information Processing Systems*, vol. 30 (Curran Associates, Inc., 2017).

26. L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "Catboost: Unbiased Boosting With Categorical Features," in *Advances in Neural Information Processing Systems*, vol. 31 (Curran Associates, Inc., 2018).

27. I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *Journal of Machine Learning Research* 3 (2003): 1157–1182.

28. G. Chandrashekar and F. Sahin, "A Survey on Feature Selection Methods," *Computers and Electrical Engineering* 40, no. 1 (2014): 16–28, https://doi.org/10.1016/j.compeleceng.2013.11.024.

29. A. Bifet and R. Gavaldà, "Learning From Time-Changing Data With Adaptive Windowing," in *Proceedings of the 2007 SIAM International Conference on Data Mining* (Society for Industrial and Applied Mathematics (SIAM), 2007), 443–448, https://doi.org/10.1137/1.9781611972771.42.

30. J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A Survey on Concept Drift Adaptation," *ACM Computing Surveys* 46, no. 4 (2014): 44.1–44.37, https://doi.org/10.1145/2523813.

31. M. D. Dikaiakos, "Grid Benchmarking: Vision, Challenges, and Current Status," *Concurrency and Computation: Practice and Experience* 19, no. 1 (2007): 89–105, https://doi.org/10.1002/cpe.1086.

32. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience* 23, no. 2 (2011): 187–198, https://doi.org/10.1002/cpe.1631.

33. V. Munhoz, V. G. Pinto, J. V. F. Lima, M. Castro, D. Cordeiro, and E. Francesquini, "Performance and Cost Evaluation of Starpu on Aws: Case Studies With Dense Linear Algebra Kernels and n-Body Simulations," *Concurrency and Computation: Practice and Experience* 38, no. 3 (2026): e70582.

34. G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conference Proceedings* 30 (1967): 483–485, https://doi.org/10.1145/1465482.1465560.

35. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM* 31, no. 5 (1988): 532–533, https://doi.org/10.1145/42411.42415.

36. P. Mattson, C. Cheng, G. Diamos, et al., "Mlperf Training Benchmark," *Proceedings of Machine Learning and Systems* 2 (2020): 336–349.

37. V. J. Reddi, C. Cheng, D. Kanter, et al., "Mlperf Inference Benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (IEEE, 2020), 446–459.

38. M. P. Chen and J. Zhu, "Deep Learning in Asset Pricing," *Management Science* 70, no. 2 (2024): 714–750, https://doi.org/10.1287/mnsc.2023.4695.

39. M. Leippold, Q. Wang, and W. Zhou, "Machine Learning in the Chinese Stock Market," *Journal of Financial Economics* 145, no. 2 (2022): 64–82, https://doi.org/10.1016/j.jfineco.2021.08.017.

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by umit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

40. J. Freyberger, A. Neuhierl, and M. Weber, "Dissecting Characteristics Nonparametrically," *Review of Financial Studies* 33, no. 5 (2020): 2326–2377.

41. S. Kozak, S. Nagel, and S. Santosh, "Shrinking the Cross-Section," *Journal of Financial Economics* 135, no. 2 (2020): 271–292, https://doi.org/10.1016/j.jfineco.2019.06.008.

42. S. Bryzgalova, M. Pelger, and J. Zhu, "Forest Through the Trees: Building Cross-Sections of Stock Returns," *Journal of Finance* 80, no. 5 (2025): 2447–2506, https://doi.org/10.1111/jofi.13477.

43. A. Y. Chen and T. Zimmermann, "Open Source Cross-Sectional Asset Pricing," *Critical Finance Review* 11, no. 2 (2022): 207–264, https://doi.org/10.1561/104.00000112.

44. T. Ribeiro, S. Singh, and C. Guestrin, "Why Should i Trust You? Explaining the Predictions of Any Classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery (ACM), 2016), 1135–1144, https://doi.org/10.1145/2939672.2939778.

45. S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in *Advances in Neural Information Processing Systems*, vol. 30 (Curran Associates, Inc., 2017).

15320634, 2026, 15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70081 by unit demirbaga - Turkey Cochrane Evidence Aid , Wiley Online Library on [28/07/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License