Primo: Practical Learning-Augmented Systems with Interpretable Models

Qinghao Hu, *Nanyang Technological University and S-Lab, NTU;* Harsha Nori, *Microsoft;* Peng Sun, *SenseTime;* Yonggang Wen and Tianwei Zhang, *Nanyang Technological University*

https://www.usenix.org/conference/atc22/presentation/hu

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

PRIMO: Practical Learning-Augmented Systems with Interpretable Models

Qinghao Hu[1,2] Harsha Nori[3] Peng Sun[4] Yonggang Wen[1] Tianwei Zhang[1]

[1]*Nanyang Technological University* [2]*S-Lab, NTU* [3]*Microsoft* [4]*SenseTime Research*

Abstract

While machine learning has demonstrated remarkable performance in various computer systems, some substantial flaws can prohibit its deployment in practice, including opaque decision processes, poor generalization and robustness, as well as exorbitant training and inference overhead. Motivated by these deficiencies, we introduce PRIMO, a unified framework for developers to design practical learning-augmented systems. Specifically, (1) PRIMO provides two interpretable models (`PrAM` and `PrDT`), as well as a *Distill Engine*, to support different system scenarios and deployment requirements. (2) It adopts *Bayes Optimization* to automatically identify the optimal model pruning strategy and hyperparameter configuration. (3) It also implements two tools, *Monotonic Constraint* and *Counterfactual Explanation*, to achieve transparent debugging and guided model adjustment. PRIMO can be applied to different types of learning-augmented systems. Evaluations on three state-of-the-art systems show that PRIMO can provide clear model interpretations, better system performance, and lower deployment costs.

1 Introduction

Over the years, machine learning (ML) has been widely adopted to optimize systems across many fields, e.g., storage [29,82,85], network [66,77,95], security [24,28,74], compiler optimization [8,93,94] and cluster scheduling [65,89,92]. These learning-augmented systems demonstrate marvelous performance compared with conventional heuristic or mathematical optimized systems.

However, most of these applied models are very complex and treated as black-boxes to developers, which brings significant gaps in deploying them in practice. **First, building a production-level learning-augmented system can incur huge costs.** From the experience at Microsoft [42], the model training process could take days to weeks with massive data. Some systems require frequent model updates to adapt to dynamic environment changes, whose cost often exceeds enterprise expectations. For some scenarios with limited data samples, developers have to use techniques to synthesize training samples [12,45,96], which inevitably introduce bias to the model and cause performance deterioration in practice [8,10,66]. Moreover, the inference process of these complicated models can pose heavy computational pressure to systems which have high real-time requirements [43,81,82], which can significantly restrict parallel capabilities and affect scalability in practice.

Second, the prediction process of these black-box models are unintelligible to humans. Developers lack understanding and trust of the model's behavior [19,53,91], which makes it difficult for them to perform model adjustments and ad hoc debugging in practical scenarios. Some efforts have been made to improve system transparency through interpreting black-box models [26,27,55]. They typically build *surrogate* models to obtain explanations for individual predictions, thus validating model behaviors and diagnosing system mistakes. However, they cannot provide an interpretation fidelity guarantee, and therefore the corresponding explanations are unreliable and potentially misleading [58,70]. In addition, they cannot address the aforementioned system cost issue.

In this paper, we aim to resolve the above challenges and facilitate *transparent, accurate* and *lightweight* system deployment in practice. We introduce PRIMO (**Pr**ior-based **i**nterpretable **m**odel **o**ptimization), the *first* unified framework that assists developers to design and optimize learning-augmented systems with interpretable models. The design of PRIMO is based on two key insights. First, *simple interpretable models have the capability of handling complex system problems*. Interpretable models do not sacrifice prediction accuracy [35,62,72], and simple model structures with low resource overhead are very suitable for real-time systems. Their effectiveness is often underestimated [70]. Second, *prior experience and domain knowledge can be leveraged by developers to further optimize the interpretable models* [20,76], which is hard to achieve for black-box models.

PRIMO makes several innovations to enhance learning-augmented systems. First, to provide comprehensive support for different systems, PRIMO introduces two interpretable model algorithms: `PrAM` is designed for better prediction accuracy and `PrDT` applies to systems with strict latency or computation constraints. PRIMO can help developers select a suitable model *automatically* based on their system requirements, including latency, accuracy, and resource budget. In addition to training models directly, PRIMO also supports distilling existing complex models, which applies to exploration-based systems with reinforcement learning (RL) [23,49,56,59].

Second, to fully exploit the potential of interpretable models, we design several built-in mechanisms to optimize model performance leveraging *prior* information. (1) PRIMO implements *Bayes Optimization* to find the optimal model pruning strategy and hyperparameter configurations for higher prediction accuracy and lower computation overhead. It fully takes advantage of prior search information to minimize the search space and training cost. (2) PRIMO also facilitates model post-processing for developers with their domain knowledge. Specifically, it provides two tools for model adjustment through adding *Monotonic Constraints* and transparent debugging with *Counterfactual Explanations*.

Based on these innovations, PRIMO provides not only precise and comprehensive interpretations for developers to understand and adjust models, but also better prediction accuracy and smaller overhead. To extensively evaluate these benefits in real scenarios, we apply PRIMO to three state-of-the-art learning-augmented systems, including two *online* systems (LinnOS for flash storage [29] and Pensieve for video steaming [51]), and an *offline* system (Clara for SmartNIC offloading [66]). For LinnOS, PRIMO provides a 2.8× system performance improvement, and reduces model training time by over 100×, as well as inference latency by over 20×. For Clara, PRIMO beats a series of black-box models in prediction accuracy and saves over 10× training cost. For Pensieve, PRIMO achieves better generalization ability and a 79× inference latency reduction. We believe PRIMO can bring similar benefits to other learning-augmented systems as well.

To summarize, we make the following contributions:

- To the best of our knowledge, PRIMO is the first framework to provide inherent interpretability for learning-augmented systems development.
- We design built-in mechanisms and adjustment tools for developers to achieve *transparent*, *accurate* and *lightweight* system deployment in practice.
- For the first time, we demonstrate that simple interpretable models can outperform complex black-box techniques in various real systems.

2 Background and Motivation

2.1 Learning-Augmented Systems

Learning-augmented systems apply machine learning techniques to optimize system performance [42]. They typically build various ML models to obtain preeminent system policies from historical execution data, such as Support Vector Machines (SVM) [65, 66], Random Forest (RF) [5, 86], Gradient Boosting Decision Tree (GBDT) [32, 92]. With the popularity of deep learning (DL) algorithms, they were also introduced to further enhance systems, e.g., Deep Neural Network (DNN) [29, 82], Convolutional Neural Network (CNN) [43, 53], Recurrent Neural Network (RNN) [66, 90] and Reinforcement Learning (RL) [41, 51]. We classify them into the following categories.

Taxonomy. Learning-augmented systems typically follow similar design workflows to integrate ML models into system operations. Based on the optimization type, they can be classified into two categories. (1) **Prediction-based** systems utilize the *supervised learning* paradigm (e.g., classification, regression) to optimize system problems. (2) **Exploration-based** systems usually adopt *reinforcement learning* to learn optimal policies in an explore-exploit way. Since there are relatively fewer unsupervised learning-based systems in practice, we consider them as our future work (§8).

Based on system requirements and application scenarios, learning-augmented systems can be divided into the following two types. (1) **Online** systems require the ML model to make prompt predictions for real-time data. Developers need to consider model inference latency and computation overhead, in addition to prediction accuracy. (2) **Offline** systems usually do not need to deploy ML models for real-time serving and have no latency or computation requirements. These systems are performance-critical and the objective of ML models is to improve prediction accuracy.

PRIMO is designed as a unified framework, providing respective optimization mechanisms for different types of learning-augmented systems.

2.2 Challenges and Motivation

While plenty of work has demonstrated the potential of ML techniques in improving system performance, there exist several challenges in the development and deployment of learning-augmented systems in practice.

Model development. First, building a qualified ML model for the target system has the following two challenges:

- **C1: high training and tuning cost.** As stated by Microsoft AutoSys [42], costs of ML model training often exceeds enterprise expectations. Real system environments are dynamically changing and stale models will cause performance deterioration. Therefore, frequent model fine-tuning or retraining is necessary, which could take days to weeks [42, 52] in order to outperform heuristic algorithms. If there are not enough GPU resources, the update time will become even more intolerable for DL models.
- **C2: susceptible to the quantity and quality of data.** A large amount of high-quality training data are essential to produce satisfactory ML models. However, in some cases, insufficient data [8, 10, 66] or excessive data collection cost [52, 88] hinder developers from training qualified models. Possible solutions include data augmentation and synthesis [12, 45, 96]. Nevertheless, owing to the sophisticated distribution of real-world data, the generated data inevitably introduce bias and shift to the learning model [66], which could compromise the system performance in practice.

Model deployment. Second, deploying ML models in practice has interpretability and inference overhead issues:

- **C3: opaque decision making process.** Developers mainly

Strategies	Interpretation Fidelity	Local Interpretation	Global Interpretation	Transparent Adjustment	Deployment Cost	Accuracy ↗	Roustness ↗	Latency ↘
Black-box models (e.g., *DNN, RL, GBDT*)	✘	✘	✘	✘	$$$	★	★	★
Interpreting black-box models [26,67]	✘	✔	✘	✘	$$$	★	★	★
Building interpretable models (PRIMO)	✔	✔	✔	✔	$	★☆	★☆	★☆

Table 1: Comparisons of different strategies for learning-augmented systems (☆: Performance improvement).

focus on improving key system metrics (e.g., I/O latency [29], user experience [51]) when designing and evaluating ML models, while ignoring their *interpretability*. As a result, most of these learning models are *black-boxes* whose prediction processes are unintelligible to humans [26, 40, 70]. Due to such opacity, system operators cannot guarantee model predictions are risk-free and have insufficient confidence to deploy them.

- **C4: difficulty in troubleshooting and adjustment.** In order to achieve expected performance in production environments, system operators typically need to adjust the learning models according to the actual scenarios [19,53,55], including input features alteration, model structure modification, data augmentation, etc. All these actions require the operators to have a profound understanding of the system and the corresponding ML technique [26, 42], which is difficult when the model is complex. In addition, ad hoc debugging is another substantial challenge to learning-augmented systems for black-box models. Improper modifications may cause severe performance degradation.

- **C5: exorbitant deployment overhead.** The model deployment overhead is another key factor for system operators' consideration [53]. The latency and computation requirements of some systems [43, 81, 82] are far more strict than conventional AI tasks. High inference overhead can cause side effects to production workloads and limit their parallelism capability [29], which can further restrict deployment scalability.

2.3 Model Interpretation as a Solution

One possible solution to address the above challenges is model interpretation. There are two primary directions to apply model interpretation for learning-augmented systems.

2.3.1 Interpreting Black-box Models.

The essential idea is to leverage existing interpretation methodologies to interpret the black-box models, making them more intelligible and transparent. A variety of interpretation tools (e.g., Lime [67], Captum [39], Shap [48]) were designed to explain the mechanisms of DNN models for CV and NLP tasks. Similar studies were also performed for other domains. For instance, Lemna [26] employs a mixture regression model [36] to interpret RNN models in DL-based security applications. Metis [55] proposes to interpret networking systems with the decision tree or hypergraph. However, we argue that the idea of interpreting black-box models is not sufficient for learning-augmented systems for the following reasons.

(1) ***No fidelity guarantee***. These tools typically interpret black-box models in a *post hoc* way, where another *local surrogate* model is created to explain the original model. They cannot have a fidelity guarantee with respect to the original model. Therefore, the corresponding explanations are often unreliable, and can be misleading [58, 70]. The fidelity of some widely applied interpretation methods (e.g., attention-based explain [84]) are still in dispute [34, 83]. Appendix B.1 presents an example of contradictory XGBoost explanations.

(2) ***Limited interpretation***. Most existing tools (e.g., Lime, Lemna) focus on explaining individual predictions (local interpretation) instead of the entire model behavior (global interpretation). Thus, the interpretation results typically cannot yield enough information for system troubleshooting. Appendix B.2 shows their insufficiency for global understanding and model surrogate.

(3) ***Requiring domain knowledge***. Different systems may employ different models and algorithms. There is no unified tool that can provide comprehensive support for interpreting arbitrary models. Consequently, domain knowledge and manual efforts are required to implement the tools and understand the explanation results. This poses a huge challenge for developers to design a learning-augmented system.

(4) ***Incapability of handling other challenges.*** Those tools only focus on model interpretation and understanding (**C3** & **C4**), but ignore other challenges discussed in §2.2.

2.3.2 Building Interpretable Models

A more promising direction, which is adopted in PRIMO, is to train *interpretable models* directly for learning-augmented systems. Interpretable models refer to the models that are inherently intelligible, where their explanations provided by themselves are faithful to what the models actually compute [58, 70]. Common interpretable models include linear regression, logistic regression, decision tree, decision list, etc. They have great potential to enhance different types of learning-augmented systems.

According to our observation from recent state-of-the-art learning-augmented systems [1], the scale of models in these systems tend to be relatively smaller than popular production-level AI models (although they are still too complex for humans to understand). For instance, the number of neurons in a RL-based system is typically less than 10K [19]. This is because most data samples in learning-augmented systems are well structured, with good representations in terms of naturally meaningful features. In such scenarios, a much sim-

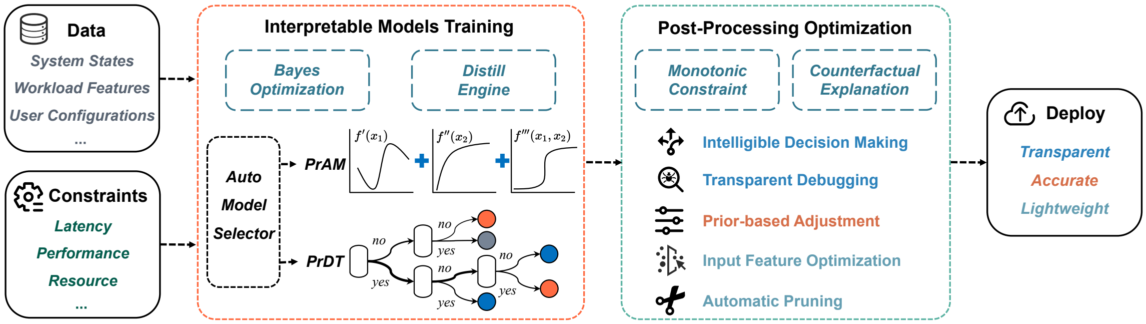


Figure 1: The workflow of learning-augmented system development using PRIMO.

pler interpretable model can give comparable performance to complex black-box models [70]. Therefore, developers can employ interpretable models for their systems, which require less data, training and tuning cost (**C1** & **C2**). The models give more information for system operators to understand (**C3**), troubleshoot and adjust (**C4**), and the inference speed is much faster than the original black-box model (**C5**).

Summary. The benefits of PRIMO compared with other methods are summarized in Table 1. It can provide not only highly precise and comprehensive interpretations for developers to understand and adjust models, but also higher accuracy and robustness, and smaller training and inference overhead. These greatly facilitate model deployment in practice.

3 PRIMO Design

We introduce PRIMO, a unified framework that assists developers to design practical learning-augmented systems. Particularly, (1) we employ *transparent* and *deterministic* interpretable models to circumvent the uncertainty issues of black-box model inference. (2) We integrate new tools for developers to leverage prior knowledge to optimize interpretable models *automatically*. (3) We design a built-in mechanism to search optimal hyperparameters in a *fast* and *convenient* way, without extra effort from the developers. Based on these designs, PRIMO can address all the challenges in §2.2.

3.1 Framework Overview

PRIMO optimizes both the *training* and *post-processing* stages of building learning-augmented systems. Figure 1 illustrates the development workflow with PRIMO. In the model training stage, PRIMO provides two interpretable model algorithms (PrAM and PrDT) designed for different system scenarios[1]. PRIMO helps developers automatically select suitable algorithms based on their system requirements including latency, accuracy, and resource budget. It supports training the interpretable model directly, or converting an existing complex black-box model into a simple interpretable model through the *Distill Engine*. We also leverage *Bayes Optimization* to find the optimal model pruning strategy and hyperparameter configurations for higher prediction accuracy and lower computation overhead. After the model is trained, PRIMO offers several optimization tools in the post-processing stage, e.g., prior-based model adjustment through adding *monotonic constraints*, transparent debugging with *counterfactual explanations*. Below we detail the mechanism of each component.

3.2 Interpretable Models

As introduced in §2.1, different system scenarios have different requirements for the learning models. To this end, PRIMO employs two types of interpretable model algorithms: `PrAM` is designed for better prediction accuracy and `PrDT` applies to systems with strict latency constraint or computation sensitivity. PRIMO supports automatic model selection based on the demands specified by the developers.

3.2.1 `PrAM`: Addictive Model based Method

Our first interpretable model, `PrAM`, is based on the Standard Generalized Additive Models (GAMs) [30]. GAMs consist of a series of *shape functions* $f_i(\cdot)$ and an intercept μ_0 (Equation 1). Since each shape function considers only one univariate term (the ith feature $\boldsymbol{x}^i$) and their combination is additive, GAMs are interpretable: we can clearly understand the contribution of each single feature to the final prediction.

Compared with linear interpretable models (e.g., logistic regression), GAMs can cope with more complex prediction tasks because shape functions are typically nonlinear and have better fitting capability. To further increase model performance, we adopt the state-of-the-art GAM algorithm: GA^{2}M [47], which additionally considers the interactions of two features and maintains the interpretability (more details are in Appendix A.1). GA^{2}M has the following form:

$$g(E[y \mid \boldsymbol{x}]) = \mu_0 + \underbrace{\sum f_i\left(\boldsymbol{x}^i\right)}_{\text{GAM}} + \underbrace{\sum f_{ij}\left(\boldsymbol{x}^i, \boldsymbol{x}^j\right)}_{\text{Interactions}} \qquad (1)$$

where $g(\cdot)$ is a link function that adapts GA^{2}M to different tasks, e.g., regression (identity), classification (logistic function); $f_{ij}(\cdot)$ represents the interaction effect of features i and j, which can be visualized as a two-dimensional heatmap.

In our implementation, `PrAM` extends the open-source library EBM [63] to obtain the optimal model with high compactness and accuracy. Compared to the complex DL models, `PrAM` can not only provide interpretability, but also takes less

[1] Other interpretable models can also be conveniently integrated into this framework, which will be considered in our future work.

training resources (without the need of GPUs) and training data samples, significantly reducing the training time and cost.

3.2.2 PrDT: Decision Tree based Method

Our second interpretable model PrDT is constructed from Decision Trees (DTs). DTs are binary tree-structured models where each *branch node* tests a condition and each *leaf node* makes a prediction [71]. Because DTs are non-parametric and can be essentially expressed as an equivalent rule list, they are transparent and simple to interpret how a prediction is obtained. Besides, the decision-making processes of DTs can be visualized so developers can easily adjust the trees according to the system requirements. They present powerful prediction capability for both classification and regression tasks, even compared with complex black-box models.

In addition to the excellent interpretability and accuracy, DTs have extremely low computation overhead and inference latency. Consequently, they are applicable to many scenarios with strict latency and resource constraints [29, 61]. Besides, DTs also exhibit other benefits, including robust performance under dynamic system environments, requiring less training data and no data preprocessing overhead during inference.

It is necessary to optimize the complexity of a DT to avoid the overfitting issue, which can affect the model generalization, accuracy and computation overhead. Instead of adding constraints (e.g., maximum depth, minimum number of samples for a leaf node) during DT training, PrDT trains a full decision tree without any limitation to capture more information from the training dataset. We adopt *minimal cost-complexity pruning* [14] to prune the full tree in the *post-processing* stage, which is elaborated in Appendix A.2.

3.3 Model Training

Primo supports two training modes. (1) *Direct*: the developer can train an interpretable model from scratch. This applies for most **prediction-based** systems. (2) *Distill*: the developer can generate an interpretable model from the original black-box model through the *Distill Engine*. This is mainly for **exploration-based** systems. To obtain high-quality models, both modes support the integration of *Bayes Optimization* for efficient model structure and hyperparameter search.

3.3.1 Bayes Optimization

There exists a trade-off between the model complexity and accuracy for both interpretable models. In order to find accurate and succinct models, Primo leverages Bayes Optimization (BO) [76], an iterative algorithm to automatically search for the optimal model configurations.

Objective function. For both PrAM and PrDT, we build a universal model scoring function $S(\theta)$ to quantify the model performance and complexity as the search objective:

$$S(\theta) = P(\theta) + \lambda \cdot C(\theta)^{\gamma} \qquad (2)$$

where $P(\theta)$ represents the model performance (e.g., classification accuracy) under hyperparameters θ during validation; λ is a knob that controls the model complexity according to users' preference; $C(\theta)$ is a metric for model complexity. For PrDT, $C(\theta^{\mathrm{PrDT}}) = N_{leaves} \times N_{depth}$, where we consider both the number of tree leaves and tree depth since unbalanced-deeper trees typically cost longer condition inference time. For PrAM, $C(\theta^{\mathrm{PrAM}}) = N_{interactions} \times N_{maxbins}$, where both the number of feature interaction terms and maximum number of bins in the feature histogram are included. Besides, the normalization factor γ regulates the effect of the model complexity.

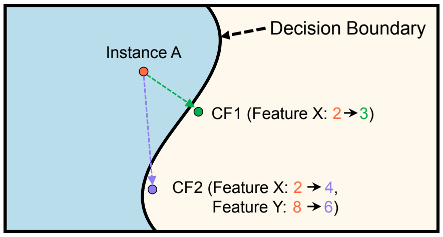


Figure 2: Illustration for the counterfactual explanation.

Prior-based hyperparameter search. Specifically, Primo employs Gaussian Process (GP) as the probabilistic surrogate model of the objective function $S(\theta)$ in Equation 2. The prediction of GP follows a normal distribution: $p(S \mid \theta, \Theta) = \mathcal{N}\left(S \mid \hat{\mu}, \hat{\sigma}^2\right)$, where Θ indicates the hyperparameter search space. To determine which point should be evaluated next, Primo adopts expected improvement (EI) as the acquisition function to trade-off exploration and exploitation [20]. In each iteration, Primo generates a set of hyperparameters and evaluates them on the interpretable model to obtain new results which are used to update the surrogate model. Compared with Grid Search (GS) and Random Search (RS) [13], BO is more efficient since it fully utilizes the prior information to minimize the search space. For instance, as shown in Figure 17 in Appendix A.2, BO can rapidly reduce the search space to a smaller size (10^{-5}~10^{-2}) for a better focus.

3.3.2 Distill Engine

In some scenarios, the learning models require special optimization. For instance, LinnOS [29] leverages *biased training* to reduce the false submit rate while causing the higher false revoke rate. Primo introduces the *Distill Engine*, which can build an interpretable surrogate model to approximate the behavior of the original black-box learning model using knowledge distillation [7, 31].

Another application of the *Distill Engine* is RL policy extraction. Both PrAM and PrDT work well for **prediction-based** systems using supervised learning, but are less supportive for **exploration-based** systems due to their incompatibility with RL. A series of works [11, 69, 75] have demonstrated the feasibility of converting NN-based learning policies to an interpretable models. Primo adopts Viper [11] to perform RL policy extraction. Specifically, we collect the trajectories of $\{\boldsymbol{s}_i, a_i\}$ pairs (i.e., system states $\boldsymbol{s}_i$ and actions a_i of learned policy $\pi(\boldsymbol{s}_i, a_i)$) generated by the original RL model and perform supervised learning to build the interpretable models.

	System Scenario	ML Algorithm	Type	Primo
LinnOS [29]	Flash Storage I/O	DNN	Online	PrDT (Direct)
Clara [66]	SmartNIC Offloading	Mixture (LSTM, GBDT, SVM)	Offline	PrAM (Direct)
Pensieve [51]	Video Streaming	RL	Online	PrDT (Distill)

Table 2: Summary of case studies for PRIMO evaluation.

To obtain a robust policy, we augment the poor-performing pairs and train the model iteratively until it is converged.

3.4 Post-Processing Optimization

After the interpretable model is built, developers can use their prior knowledge to further optimize the model and enhance the system performance. PRIMO designs two tools to assist developers in model post-processing. Note that these operations are *optional* since generally the trained interpretable models already achieve satisfactory performance.

3.4.1 Monotonic Constraint

In many learning-augmented systems, the input features exhibit a monotonic relationship with the output values (e.g., higher video bitrate selection with better bandwidth). But the corresponding model often presents a non-monotonic pattern due to the sub-optimal construction strategy or noisy training data (e.g., outlier data points, biased synthetic data). This can lead to unstable performance and intelligibility degradation in practice. To this end, PRIMO leverages a method from DP-EBM [62], which adds monotonic constraints to boosted trees via post-processing. Specifically, we model this task as an isotonic regression problem [15] with respect to a complete order. The objective is to minimize $\sum_i w_i (y_i - \hat{y}_i)^2$ subject to $\hat{y}_i \leq \hat{y}_j$ and weights w_i are strictly positive. We adopt the Pool Adjacent Violators (PAV) [6] algorithm to obtain an optimal solution maintaining monotonicity, and use it to replace the original shape function of PrAM. Our tool only needs developers to provide the feature name or index and the subsequent model adjustment process is transparent and automatic.

3.4.2 Counterfactual Explanation

To make modifications to the models, developers need to answer some challenging questions, e.g., which feature related shape function should be adjusted? how to determine the modification degree? To help them make reasonable decisions, we design the Counterfactual Explanation tool in PRIMO to generate additional insights for model adjustment. As illustrated in Figure 2, this tool aims to find smaller change (green arrow) to the feature values that can alter the prediction to a predefined output within the dataset. It typically uses the k-nearest neighbors (kNN) algorithm to find k training instances with the minimum L_2 distances [80]. To address the inefficiency of the brute-force kNN approach, we propose to use Ball Tree [22] to partition data in a series of nesting hyper-spheres, thus the distance between a prediction point and the centroid is sufficient to determine a lower and upper bound on the distance to all points within the hyper-sphere node. This approach considerably reduces the query time when dealing with large-scale and high-dimensional datasets. And developers could perform guided model adjustment easily.

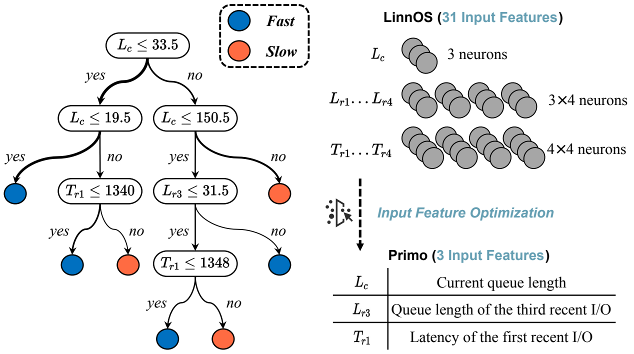


Figure 3: (**Left**) Learned PrDT model for an SSD. The thicker arrow line denotes the higher frequency. (**Right**) PRIMO optimizes the input features of LinnOS. Each feature represents a *digit* in LinnOS while a complete *number* in PRIMO.

PRIMO *Experiments.* *In the following three sections, we will present three case studies to demonstrate how* PRIMO *can optimize state-of-the-art learning-augmented systems. Table 2 describes these three scenarios. The key observation for each case is summarized in Appendix C. We believe* PRIMO *can be applied to other learning-augmented systems as well.*

4 Case Study 1: LinnOS

As the first case, we consider LinnOS [29], a learning-based operating system that accelerates storage applications. LinnOS adopts a 3-layer neural network (31-256-2, in total of 8706 parameters) for *each* SSD to precisely predict its performance. To achieve this, it collects the traces of real workloads running on the SSD and obtains fine-grained information (per I/O), including recent queue lengths and latency. Instead of predicting the concrete latency values, LinnOS simplifies it as a binary (fast / slow) classification task through setting an inflection point (*IP*). More details about LinnOS and our implementation can be found in Appendix D.1.

PRIMO automatically selects the PrDT model for LinnOS, since it has comparable accuracy and lower inference latency than PrAM. For comprehensive evaluation, we consider two models with different optimization objectives: *efficiency*-oriented (PRIMO-E) and *performance*-oriented (PRIMO-P). We compare PRIMO with two baselines. (1) Base: the vanilla Linux I/O mechanism. (2) LinnOS: we set the inflection point of LinnOS as a constant percentile (at p85 latency) and apply the biased loss to the model training (all keep the same).

4.1 System Interpretation

The primary goal of PRIMO is to provide interpretation for the target system. Figure 3 (left) presents the learned decision tree (PRIMO-E) for one SSD. The explanation of each notation can be found on the right side. From this tree, we can clearly understand how PRIMO makes decisions for each prediction

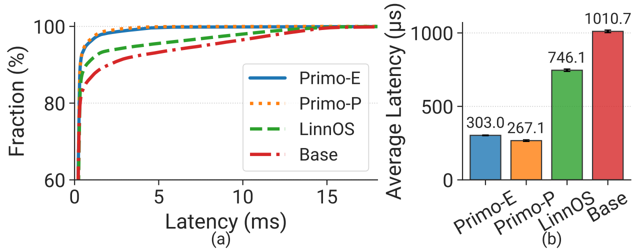


Figure 4: Overall performance comparisons. (a) CDF of I/O latency. (b) Average I/O latency.

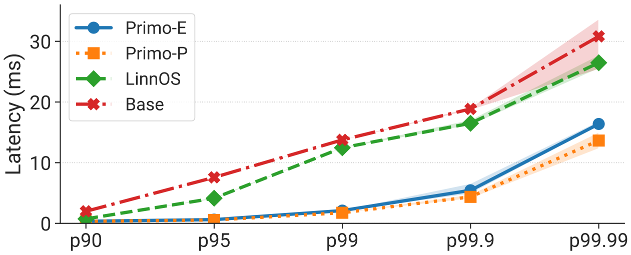


Figure 5: Tail percentiles of I/O latency.

(*Local interpretation*). We can also obtain intuitive cognition of the overall model behavior (*Global interpretation*) through observing the thickness of each decision path (arrow lines).

Specifically, the top-2 layers of the DT show PRIMO first classifies I/O requests from the *current queue length* (L_c), indicating this feature can significantly affect the prediction results. Developers can perform adjustments to L_c thresholds to optimize system behavior. Because the 4-layer DT only contains 7 leaves (terminal nodes), each prediction needs to take at most 4 condition tests at the branch nodes and the majority of test instances only need to execute 2 condition tests. This inference overhead is much smaller than the original DNN model with 8706 parameters in LinnOS. Moreover, as shown in Figure 3 (right), PRIMO only takes 3 input features without any preprocessing, which further reduces the model complexity and deployment overhead. On the contrary, LinnOS needs to perform input data preprocessing for all 9 metrics to form a 31-dimensional input feature (e.g., $L_c = 15$ needs to be converted into a $\{0, 1, 5\}$ vector). This operation is necessary for *every* I/O read operation, remarkably exacerbating the inference overhead.

4.2 Performance Analysis

We evaluate the performance of PRIMO in the LinnOS flash storage I/O scenario from the following two perspectives:

Overall performance. Figure 4 shows the Cumulative Distribution Function (CDF) and average I/O latency (with the standard deviation) of each method over three independent experiments. It is obvious that both PRIMO-E and PRIMO-P significantly outperform LinnOS. Compared with the base I/O mechanism, LinnOS reduces 26.2% I/O latency on average, while PRIMO decreases the I/O latency by 70.0~73.6%. It indicates PRIMO can achieve an additional 2.5~2.8× (PRIMO-E / PRIMO-P) improvement over LinnOS.

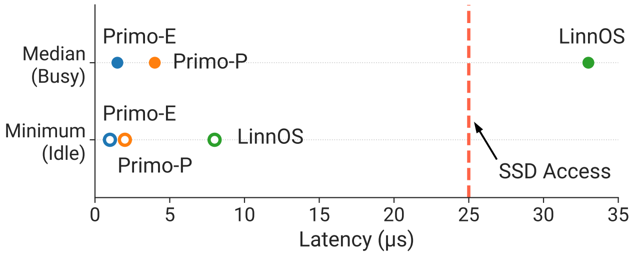


Figure 6: Model inference latency. Empty circles represent the minimum inference latency when the system is idle. Solid circles represent the inference latency of the median I/O operation when the system is busy. The vertical line indicates the basic SSD access latency (reading 4KB data in the idle state).

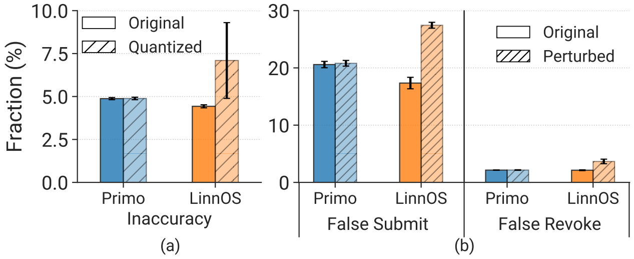


Figure 7: (a) Quantization impact. (b) Robustness test.

Tail performance. The tail behavior is critical to system performance. Figure 5 presents the average I/O latency and the range at tail percentiles (from p90 to p99.99). We find LinnOS fails to reduce tail latency on the tail, and the curve almost overlaps with the Base case. On the contrary, PRIMO-P achieves 7.9×, 4.3× and 2.3× performance improvement over the vanilla I/O mechanism at p99, p99.9 and p99.99 respectively. Additionally, PRIMO-P also performs much better at p90 (2.2×) and p95 (7.5×) compared to LinnOS.

4.3 Effectiveness Analysis

We perform the effectiveness analysis from the following perspectives to investigate why PRIMO can outperform LinnOS.

Inference overhead. In Figure 6, we measure the extra inference latency of PRIMO and LinnOS. (1) When the system is idle, we measure the minimum inference latency. We observe that LinnOS takes $8\mu s$, while the overhead of PRIMO-E is almost negligible ($\leq 1\mu s$), making the deployment more lightweight. (2) When the system is busy with heavy I/O operations, LinnOS requires a median inference latency of $33\mu s$ due to the high frequency of preprocessing and inference. This is even higher than the basic SSD access latency ($25\mu s$). In contrast, PRIMO remains relatively lower inference latency with smaller overhead.

Quantization. Since floating points are not well supported in the Linux kernel, the model weights of LinnOS and the thresholds of PRIMO are converted to integers by quantization. This can achieve smaller inference latency at the cost of accuracy degradation. Figure 7 (a) shows the quantization

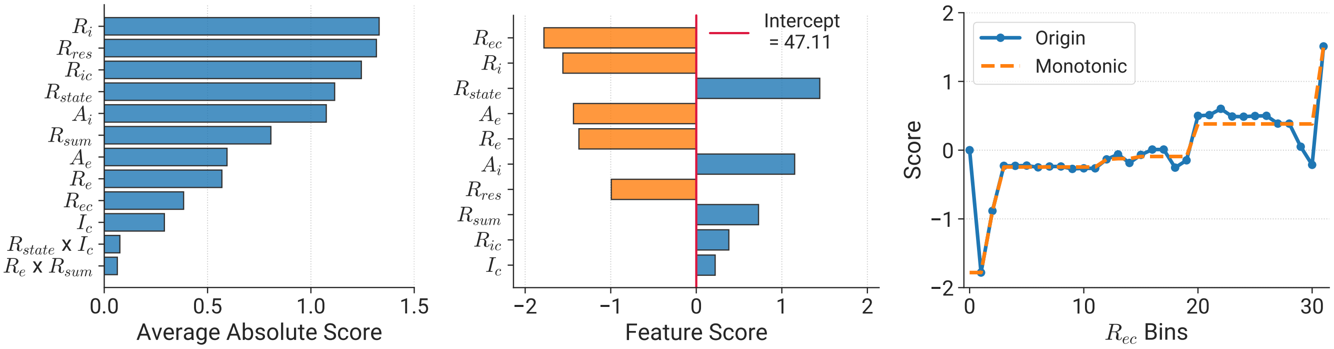


Figure 8: Interpretation and visualization of the PrAM model in Clara-MS. (**Left**): Global interpretation of overall feature importance. (**Middle**): Local interpretation of each feature's contribution to individual predictions. (**Right**): Visualization of the learned shape function of R_{ec} (blue line), and with the monotonic constraint post-processing optimization (orange line).

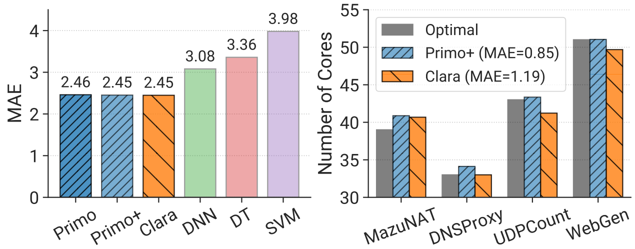


Figure 9: Evaluation on Clara-MS. (**Left**): Mean Absolute Error (MAE) of testset. (**Right**): Prediction of 4 real NFs.

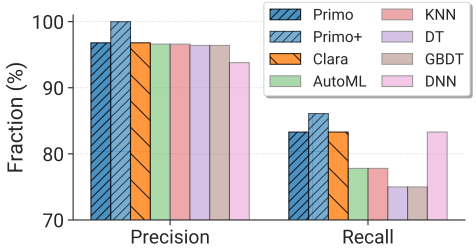


Figure 10: Model precision and recall rates in Clara-AI.

impact on the prediction accuracy. It is evident that the accuracy drop of LinnOS is over 2% and varies significantly among different SSD models. In comparison, PRIMO-E has negligible accuracy degradation, as the node threshold values are naturally integers or the decimal part is 0.5.

Robustness. A good model should exhibit high robustness against system state drifting. To measure the robustness of those methods, we synthesize some perturbed samples by adding Gaussian noise to the test dataset. The noise is added to all 4 recent I/O queue lengths ($\sigma = 5$) and I/O latency ($\sigma = 100$)[2]. Figure 7 (b) illustrates the false submit and false revoke rates of LinnOS and PRIMO-E under the original and perturbed test datasets. Reducing the false submit rate is far more important since the *failover* overhead of false revoke is negligible. It is obvious that PRIMO keeps stable accuracy under perturbed input while LinnOS presents severe performance degradation. The robustness of the interpretable model in PRIMO derives from fewer input features and the inherent stability of the tree structure compared with the DNN model.

5 Case Study 2: Clara

Clara [66] is an offline tool that generates offloading insights for network functions (NFs) on SmartNICs. It can analyze a legacy NF in its unported form and suggest the optimal offloading strategies. The main challenge of adopting those ML techniques in Clara is that *insufficient* SmartNIC programs can be served to produce training data. Clara has to utilize YarpGen [45] to generate abundant synthesized programs. Clara contains several components for the generation of different offloading insights. Each component adopts a ML algorithm as described below:

- **Multicore Scale-out analysis (Clara-MS).** SmartNICs use multicore parallelism to improve packet processing performance. Clara adopts *GBDT* [17] to predict the optimal number of cores for each NF.
- **Algorithm Identification (Clara-AI).** Certain packet processing algorithms in the host NF can benefit from ASIC accelerators in the SmartNIC. Clara adopts *SVM* [73] to identify such code blocks.
- **Cross-platform Prediction (Clara-CP).** Clara trains an *LSTM* network [9] to predict the number of compute and memory instructions that a NF can be compiled to.

We employ the PrAM model to replace all the three ML models in Clara, as it has better accuracy than PrDT. To analyze the effectiveness of transparent model adjustment in PRIMO, we also evaluate the model performance with post adjustment (denote as PRIMO+). We compare PRIMO with the original models in Clara (LSTM, GBDT and SVM), as well as some alternative baseline algorithms (CNN, DT, TPOT [64] (namely AutoML), K-Nearest Neighbor (kNN)). More details about Clara and our implementation can be found in Appendix D.2.

5.1 System Interpretation

As shown in Figure 8, PRIMO provides comprehensive interpretation for the Clara-MS task, including global and local

[2] Since the current queue length L_c is the most significant feature, we do not modify its value to avoid changing the real label.

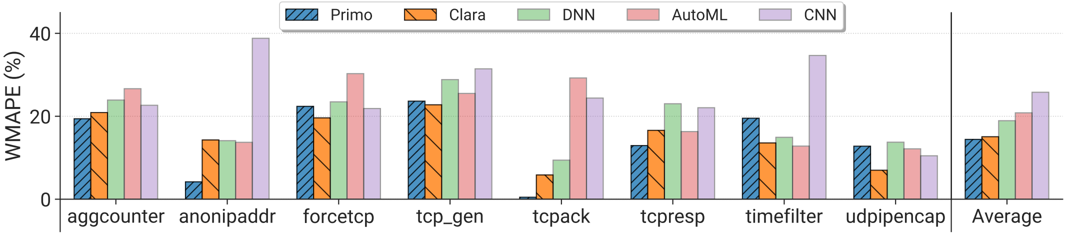


Figure 11: Weighted mean-absolute percentage error (WMAPE) over 8 types of NFs in Clara-CP.

interpretation, as well as transparent shape functions. We list the notation descriptions in Appendix D.2. From the left figure, we find R_i, R_{res} and R_{ic} are the most important features that contribute most to model prediction. Developers should pay more attention to shape function optimizations for these features. We also notice the impact of feature interactions is relatively less important, indicating that we can reduce their priority in model optimization. The middle figure presents the interpretation of the individual prediction for *UDPCount* NF. The final prediction equals the sum of every feature score and the intercept constant (Equation 1). Through the local interpretation, developers can clearly check the model behavior for each prediction to make the corresponding adjustment. Moreover, the right figure (blue line) illustrates the learned shape function for R_{ec}, which allows developers to dive deeper into fine-grained model adjustment (such as the orange line).

5.2 Performance Analysis

Since Clara is an offline system, for each task, we mainly evaluate the model accuracy rather than the inference cost.

Clara-MS. As shown in Figure 9 (left), our interpretable model in PRIMO achieves similar accuracy as the GBDT (XGBoost [17]) model in Clara, and outperforms other ML models over the synthesized test dataset. Figure 9 (right) further presents the accuracy of PRIMO for 4 real NFs. Compared to Clara, PRIMO achieves 1.4× less prediction errors and at most 5% error to the optimal configurations.

Clara-AI. In Figure 10, PRIMO achieves the equivalent precision and recall rates as the SVM model in Clara, and beats other ML algorithms. Through successfully identifying CRC-based NFs, PRIMO could improve peak throughput by 1.6× and decrease latency by 25% [66].

Clara-CP. Clara uses the LSTM model to predict the number of instructions for unported codes. Figure 11 shows the accuracy of the Clara-CP task over 8 representative real NFs and the `Average` column represents the WMAPE results across all the NFs. We observe PRIMO (14.4%) delivers better performance than Clara (15.1%). This demonstrates the capability of PRIMO to cope with complex program embeddings.

5.3 Model Adjustment

To overcome the training data insufficiency issue, Clara uses YarpGen [45] to generate synthesized programs. This inevitably introduces certain data distribution drifts from the actual scenario. Specifically, there exist instruction distribution differences (0.0303 of Jensen-Shannon divergence and 0.0354 of Bhattacharyya distance) between real-world and synthesized click programs [66]. Such drifts could compromise the model performance. The transparency of the PRIMO model allows developers to discover and fix undesirable behaviors caused by the synthesized data. In addition, PRIMO designs two post-processing tools to help developers adjust the models based on their domain knowledge:

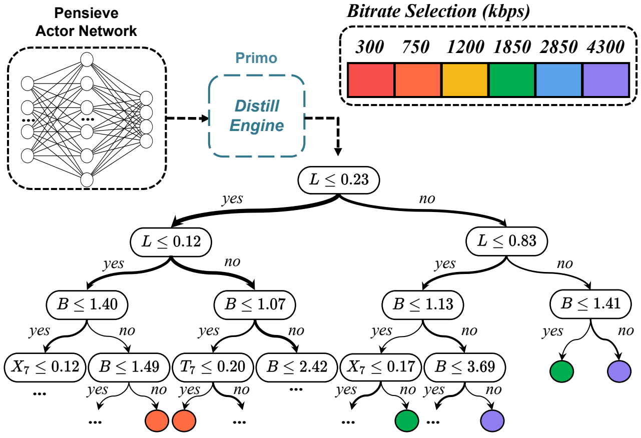
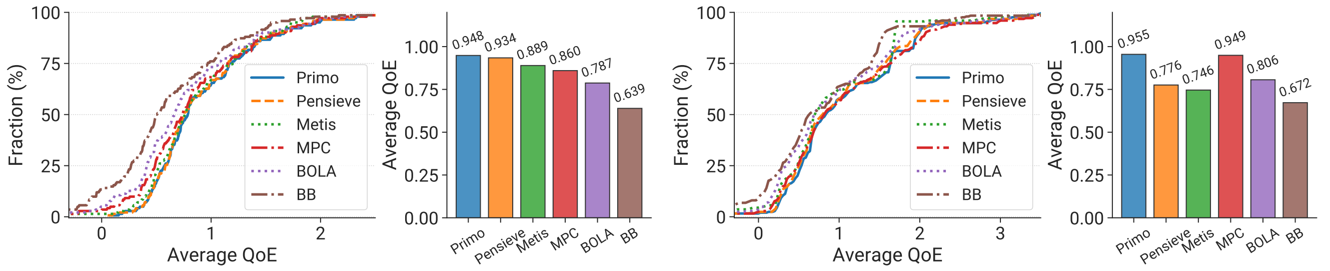


Figure 12: Visualization of the interpretable model distilled from the Pensieve policy. For simplicity, we only present the top 5 layers, and the ellipsis indicates subsequent nodes.

Monotonic Constraint. As introduced in §3.4.1, developers can leverage PRIMO to generate a new shape function with monotonic constraint and rectify the incorrect behaviors of the models automatically. For instance, in Figure 8 (right), the developers know the desired number of cores should be proportional to the memory/compute intensity, i.e., R_{ec} (EMEM/Compute Ratio). Then they can replace the original shape function (blue line) with the monotonic shape function (orange line). They can check each shape function and decide whether it is necessary to apply such adjustment based on their prior knowledge. To evaluate the effectiveness of this strategy, we apply the *Monotonic Constraint* tool to two shape functions (A_i & R_{ec}) and yield the adjusted model PRIMO+. As shown in Figures 9, PRIMO+ achieves better prediction accuracy. This shows the monotonicity of the PRIMO model can be achieved via simple post-processing and appropriate adjustment can bring better performance.



(a) Norway HSDPA

(b) FCC Broadband

Figure 13: Overall performance of PRIMO compared with other methods on the Norway HSDPA and FCC Broadband traces.

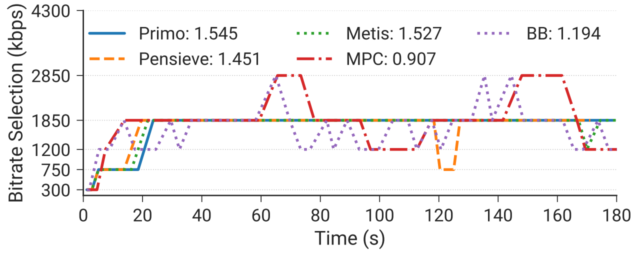


Figure 14: Profiling the bitrate selections of ABR algorithms over one typical Norway HSDPA trace. Legend presents the average QoE of each algorithm.

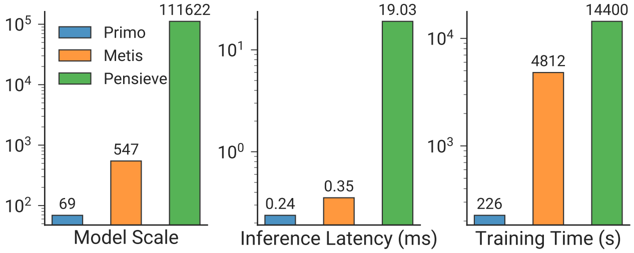


Figure 15: Comparing three learning-based ABR methods.

Counterfactual Explanation. This tool aims to provide simple and intuitive explanations for model troubleshooting. More concretely, it helps developers to understand why this prediction is wrong and how to adjust the model to fix it. We use Clara-AI as an example to describe its usage and evaluation. Clara employs Sequential Pattern Extraction (SPE) [21] to extract code features as *boolean* sequences (each containing 102 features) to indicate whether the NF program contains code blocks for acceleration. Our PRIMO model allocates a contribution score for each feature. To fix False Negative (FN) predictions, we utilize this tool to find the closest *k* instances from the data set with the opposite label. Through the comparison of these instances, we can easily discover the feature with inadvisable learned scores and adjust the score. In this case, we increase the contribution weight of the 84*th* feature appropriately. We can also perform transparent debugging for False Positive (FP) predictions similarly. In Figure 10, PRIMO+ further enhances the F1 score from 89.6% to 92.5%.

6 Case Study 3: Pensieve

Our third case study is Pensieve [51], a system that uses RL for online video streaming. It learns the adaptive bitrate (ABR) algorithms automatically to optimize the user quality of experience (*QoE*) defined in Equation 4 in Appendix D.3.

We obtain an interpretable PrDT model through distilling from the original RL actor model. Then we implement the PrDT model into the ABRController of dash.js [2]. More details can be found in Appendix D.3. For baselines, we compare PRIMO with the following algorithms: (1) The RL model in Pensieve. (2) Buffer-Based(BB) [33]: selecting bitrates with the goal of keeping the buffer occupancy above 5 seconds. (3) BOLA [78]: selecting bitrates with Lyapunov optimization on buffer occupancy observations. (4) MPC [87]: selecting bitrates with a control-theoretic model. We evaluate robustMPC variant which can better handle errors in throughput prediction. (5) Metis [55]: using a decision tree to explain the Pensieve RL model, which represents the handcrafted DT approach. Evaluations are performed on the simulator provided by Pensieve, except the deployment experiment (latency).

6.1 System Interpretation

Figure 12 illustrates the learning process with the Distill Engine, as well as the decision making process of the interpretable policy. Related notations are described in Appendix D.3. This DT contains 8 layers and 35 leaves in total, which is compact and simple enough for developers to understand its complete operation logic.

Similar to the PRIMO model in LinnOS (Figure 3), the first 2 layers divide decision flows based on the feature *L* (Last chunk bitrate) which is in line with our perception. In the third layer, PRIMO proceeds to classify environment states (inputs) according to the feature *B* (Current buffer size). These observations indicate both *L* and *B* are the key features that affect the final bitrate decision, inspiring developers to pay more attention to them when designing ABR algorithms.

6.2 Performance Analysis

Overall performance. Since the ABR algorithm could encounter unprecedented network conditions by different clients, it is important to evaluate its generalization ability. So in addition to the *Norway HSDPA* trace used for model training, we also evaluate another trace *FCC Broadband* that is never applied for training. Figure 13 presents the QoE distribution and average QoE of each method on the two traces. For Nor-

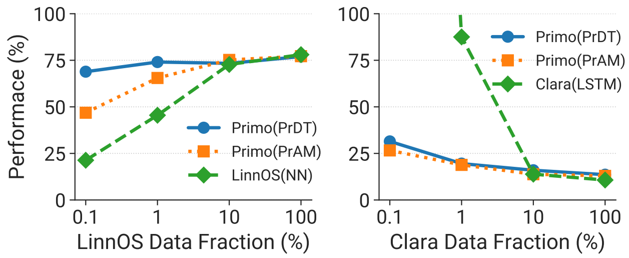


Figure 16: Model performance with less training data. (**Left**): Recall rate in LinnOS (the higher the better). (**Right**): WMAPE in Clara-CP (the lower the better).

way HSDPA, the CDF curve of PRIMO almost overlaps with Pensieve's curve, and the average QoE is even 1.5% higher than Pensieve. This demonstrates PRIMO has successfully learned the Pensieve policy with a simple decision tree and outperforms other ABR algorithms. Furthermore, for FCC Broadband, PRIMO presents better generalization than other two learning-based algorithms. Such advantages are attributed to the adaptive pruning strategy in Bayes Optimization, and the imitation process in the Distill Engine. In contrast, although Metis also uses a decision tree to get a surrogate model from Pensieve, it has some performance degradation as its inflexible pruning strategy.

Example analysis. Figure 14 profiles the bitrate selection actions of different ABR algorithms over a single network trace. We find two heuristic algorithms (BB and MPC) keep fluctuating during the video streaming, which could cause a terrible user experience. The other three learning-based algorithms have more stable decisions. Pensieve decides to decrease the bitrate at 120s and Metis chooses to reduce the video resolution at 170s. This can cause an unsmooth experience (as the penalty term in Equation 4) even though they adjust back the bitrate quickly. In contrast, PRIMO gives a much more smooth experience.

Training and inference overhead. The PRIMO model is more compact and simpler. Figure 15 (left) compares the model complexity[3] of different methods. We observe that PRIMO reduces the model scale by 1617× compared to the original Pensieve actor model. Even for Metis which also uses a decision tree, PRIMO can reduce the tree complexity by 7.9×. For inference, PRIMO only needs to perform 3~7 condition tests to make a bitrate decision. It can reduce 70× and 1.5× inference latency compared with Pensieve and Metis respectively, as shown in Figure 15 (middle). To generate a model, in Figure 15 (right), PRIMO only needs less than 4 minutes for model distillation, which is 21.3× faster than Metis (under the same setting). Compare with Pensieve 4 hours training time, less than 4 minutes distill time is ignorable. In summary, PRIMO can greatly reduce overall operating costs in the video streaming scenario.

Task	DL Model	Origin	PRIMO	Improvement
LinnOS	3 × DNN (50 epoch)	564s	5s	112.8×
Clara-CP	LSTM (30 epoch)	1,081s	79s	13.7×

Table 3: Training time comparison with original DL models.

Task	Metric	PRIMO w/o BO	PRIMO w/ BO	Improvement
LinnOS	F1 Score	0.8089	0.8518	5.3%
Clara-CP	WMAPE	0.1728	0.1442	16.6%
Clara-MS	MAE	1.0155	0.8660	14.7%

Table 4: Ablation study for Bayes Optimization.

7 More Evaluation

We run some experiments to further evaluate the benefits of PRIMO more comprehensively.

Requiring less training data. Due to the simpler model structure and fewer parameters, PRIMO can have better performance in some scenarios without enough training data like Clara. In Figure 16, we compare the performance of two PRIMO models with the original DL models in LinnOS and Clara-CP using less training data. We use a smaller dataset (10%) in LinnOS as the baseline. Because LinnOS provides abundant data for DNN model training, 10% of original data can provide equivalent performance. It is clear that PRIMO models maintain better performance with limited training data, especially for the PrDT model. Conversely, the original DL models only work with abundant data. This shows PRIMO has broader applicability for various scenarios.

Short training time. Table 3 presents the training time of PRIMO and original DL models in LinnOS and Clara-CP, which adopt the default numbers of training epochs in their papers. PRIMO is able to reduce 2-3 orders of magnitude of training time. Even considering the hyperparameter search process, the significant time conservation could maintain since multiple trails can be executed concurrently. Note that GPUs can only provide very limited acceleration (<1.2×) for these two DL models. Additionally, LinnOS requires training a DNN model for *each* SSD and the prototype only considers three SSDs. In a production-level distributed storage system with thousands of SSDs, LinnOS could have a severe scalability issue. In contrast, PRIMO remarkably saves the training cost, making the deployment more feasible in practice.

Impact of BO. We further perform an ablation study on Bayes Optimization in PRIMO. Table 4 summaries the performance of the PRIMO model with and without BO in LinnOS and Clara. We observe 5.3%~16.6% accuracy improvement brought by BO. Besides, BO typically simplifies the model scale while obtaining better performance. For LinnOS, BO reduces over 15× tree nodes compared with PRIMO without BO. This verifies the importance of the BO component in making interpretable models more practical. For search time aspect, BO obtains over 1.2× acceleration compared with naive random search algorithm in Clara-MS task by reducing search trails to reach equivalent performance.

[3] Model complexity refers to the number of parameters for Pensieve model, or number of nodes for PRIMO and Metis.

8 Discussion

Is the interpretation always correct? Yes. PRIMO provides the interpretation correctness guarantee for each generated model and each prediction. Existing interpretation tools [39, 48, 67] aim to offer explanations for understanding black-box models, whereas the generated interpretations are sometimes contradictory or even mislead users. In contrast, PRIMO models are inherently interpretable and developers can totally trust the interpretation.

Can PRIMO be applied to all systems? PRIMO has its limitations in some system scenarios. For instance, it does not yet support unsupervised learning scenarios (e.g., anomaly detection in security applications [16]). It cannot outperform black-box models in systems with extremely complex features, e.g., images, speeches. These will be our future work.

Is the post-processing step necessary? These operations are optional because the trained models without post-processing usually have excellent performance. In order to take full advantage of the interpretable models, the post-processing tools help developers leverage their expertise and domain knowledge to further optimize system performance. In a black-box model, it is hard to perform such optimization.

Can PRIMO work on a larger model? Yes. We have demonstrated PRIMO can outperform DNN models in various scenarios, including LinnOS (MLP with 8×10^3 parameters), Clara-CP (LSTM+FC with 4×10^4 parameters) and Pensieve (CNN+MLP with 1×10^5 parameters). They represent most model scale of learning augmented systems listed in [1]. For larger models, we evaluate Habitat [88] as an example. It leverages 8-layer MLP models, containing over 8×10^6 parameters, to prediction DL operation execution time on heterogeneous GPUs. PRIMO can provide comparable prediction accuracy as Habitat across `conv2d`, `linear`, `lstm` and `bmm` operations.

How to interpret high-dimensional data? Admittedly, when handling high-dimensional datasets, PRIMO models may become more complicated for users to understand the whole model. However, PRIMO provides ordered feature importance for interpretation. Generally, users can focus on the top several tree layers of `PrDT` or several significant shape functions according to the global interpretation of `PrAM`.

Future work. There are four possible directions as our future work. (1) We can extend PRIMO to support learning-augmented systems with unsupervised learning. (2) To obtain a more accurate interpretable model, we can optimize the model training algorithm. Currently, we use CART [14], the most popular and widely applied approach, for decision tree learning in `PrDT`. This is based on the heuristic greedy algorithm where locally optimal decisions are made in each node. In the future, we plan to employ novel decision tree training algorithms (e.g., GOSDT [44], based on dynamic programming method) to solve the sub-optimal problem. (3) For practical deployment, comprehensive programming language support is needed because different systems have their own coding requirements. `PrDT` has a tool for converting Python-based models to other formats. But `PrAM` only supports the conversion to the ONNX [4] format currently. We aim to provide more model format conversion in the future. (4) Integration with existing RL-based system development frameworks (e.g., Park [50]) to facilitate more practical system deployment.

9 Related Work

Interpretability of learning-augmented systems. To our best knowledge, there is no prior work that develops a unified framework for providing inherent interpretability for systems like PRIMO. Besides, interpretability is often overlooked during the development of learning-augmented systems, and only a few works consider it. Bao [53] is a learning-based system that adopts TreeCNN [60] for query optimization and the decision process can be inspected by developers. Tang et al. [79] proposed an interpretable method that extracts a Finite State Machine from a RL policy for storage resource allocation in Huawei. Grüner et al. [25] generated concise and interpretable rule-sets for unknown proprietary streaming algorithms (e.g. ABR approaches in Youtube, Twitch), which is similar to the Pensieve case study with PRIMO (§6).

Some efforts were also made on building tools for interpreting black-box models [26, 27, 55], as discussed in §2.3.1. Different from these methods, PRIMO does not seek for interpreting black-box models but directly building transparent models, with higher fidelity and efficiency.

Machine learning and system co-design. It is non-trivial to apply ML techniques for system design and deployment in practice. Autosys [42] introduces a framework to address common design considerations (e.g., learning-induced system failures, extensibility), and reported years of experiences in designing and operating learning-augmented systems at Microsoft. WhiRL [19] facilitates the safe deployment of RL-based systems through verifying whether the learned policy meets the designer's requirements. Some components in PRIMO are inspired from these works.

10 Conclusion

This work introduces PRIMO, a unified framework that assists developers to design practical learning-augmented systems with interpretable models. For different scenarios, PRIMO provides respective models and optimization solutions to meet the system requirements. Based on our case studies, we demonstrate that PRIMO can achieve *transparent*, *accurate* and *lightweight* system deployment in practice.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable comments and suggestions. This study is supported under the RIE2020 Industry Alignment Fund–Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contributions from the industry partner(s).

References

[1] Awesome-ml-for-system. https://github.com/S-Lab-System-Group/Awesome-ML-for-System, 2022.

[2] Dash.js: Javascript player. https://github.com/Dash-Industry-Forum/dash.js, 2022.

[3] Federal communications commission. https://www.fcc.gov/reports-research/reports/measuring-broadband-america, 2022.

[4] Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2022.

[5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. OSDI '20.

[6] Miriam Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid, and Edward Silverman. An empirical distribution function for sampling with incomplete information. *The Annals of Mathematical Statistics*, 1955.

[7] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? NeurIPS '14.

[8] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. A deep learning based cost model for automatic code optimization. MLSys '21.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. ICLR '15.

[10] Erick Carvajal Barboza, Sara Jacob, Mahesh Ketkar, Michael Kishinevsky, Paul Gratz, and Jiang Hu. Automatic microprocessor performance bug detection. HPCA '21.

[11] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. NeurIPS '18.

[12] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. Generating complex, realistic cloud workloads using recurrent neural networks. SOSP '21.

[13] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012.

[14] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. Classification and regression trees. 1984.

[15] Nilotpal Chakravarti. Isotonic median regression: A linear programming approach. *Mathematics of Operations Research*, 1989.

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 2009.

[17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. KDD '16.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. USENIX ATC '19.

[19] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. Verifying learning-augmented systems. SIGCOMM '21.

[20] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. ICML '18.

[21] Yujie Fan, Yanfang Ye, and Lifei Chen. Malicious sequential pattern mining for automatic malware detection. *Expert Systems with Applications*, 2016.

[22] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithms for computing k-nearest neighbors. *IEEE Transactions on Computers*, 1975.

[23] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. ICML '18.

[24] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu. Experiences of landing machine learning onto market-scale mobile malware detection. EuroSys '20.

[25] Maximilian Grüner, Melissa Licciardello, and Ankit Singla. Reconstructing proprietary video streaming algorithms. USENIX ATC '20.

[26] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. CCS '18.

[27] Dongqi Han, Zhiliang Wang, Wenqi Chen, Ying Zhong, Su Wang, Han Zhang, Jiahai Yang, Xingang Shi, and Xia Yin. Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications. CCS '21.

[28] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo Seltzer, and Haifeng Chen. SIGL: Securing software installations through deep graph learning. USENIX Security '21.

[29] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. OSDI '20.

[30] Trevor Hastie and Robert Tibshirani. Generalized additive models: Some applications. *Journal of the American Statistical Association*, 1987.

[31] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *CoRR*, 2015.

[32] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. SC '21.

[33] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. SIGCOMM '14.

[34] Sarthak Jain and Byron C. Wallace. Attention is not explanation. NAACL '19.

[35] José Jiménez-Luna, Francesca Grisoni, and Gisbert Schneider. Drug discovery with explainable artificial intelligence. *Nature Machine Intelligence*, 2020.

[36] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the em algorithm. IJCNN '93.

[37] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. NeurIPS '17.

[38] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. USENIX ATC '20.

[39] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. Captum: A unified and generic model interpretability library for pytorch. *CoRR*, 2020.

[40] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. ICML '21.

[41] Xu Li, Feilong Tang, Jiacheng Liu, Laurence T. Yang, Luoyi Fu, and Long Chen. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. USENIX ATC '21.

[42] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, and Wenjun Dai. Autosys: The design and operation of learning-augmented systems. USENIX ATC '20.

[43] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. USENIX ATC '19.

[44] Jimmy Lin, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer. Generalized and scalable optimal sparse decision trees. ICML '20.

[45] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 2020.

[46] Yin Lou, Rich Caruana, and Johannes Gehrke. Intelligible models for classification and regression. KDD '12.

[47] Yin Lou, Rich Caruana, Johannes Gehrke, and Giles Hooker. Accurate intelligible models with pairwise interactions. KDD '13.

[48] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. NeurIPS '17.

[49] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys Tutorials*, 2019.

[50] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Bojja Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. NeurIPS '19.

[51] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. SIGCOMM '17.

[52] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. SIGCOMM '19.

[53] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. SIGMOD '21.

[54] Zili Meng, Yaning Guo, Yixin Shen, Jing Chen, Chao Zhou, Minhu Wang, Jia Zhang, Mingwei Xu, Chen Sun, and Hongxin Hu. Practically deploying heavyweight adaptive bitrate algorithms with teacher-student learning. *IEEE/ACM Transactions on Networking*, 2021.

[55] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. SIGCOMM '20.

[56] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. ICML'17.

[57] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. ICML '16.

[58] Christoph Molnar. *Interpretable Machine Learning*. 2019.

[59] Shanka Subhra Mondal, Nikhil Sheoran, and Subrata Mitra. Scheduling of time-varying workloads using reinforcement learning. AAAI '21.

[60] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. AAAI '16.

[61] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. EuroSys '21.

[62] Harsha Nori, Rich Caruana, Zhiqi Bu, Judy Hanwen Shen, and Janardhan Kulkarni. Accuracy, interpretability, and differential privacy via explainable boosting. ICML '21.

[63] Harsha Nori, Samuel Jenkins, Paul Koch, and Rich Caruana. Interpretml: A unified framework for machine learning interpretability. *CoRR*, 2019.

[64] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. GECCO '16.

[65] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. OSDI '20.

[66] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. SOSP '21.

[67] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. KDD '16.

[68] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute path bandwidth traces from 3g networks: analysis and applications. MMSys '13.

[69] Aaron M. Roth, Nicholay Topin, Pooyan Jamshidi, and Manuela Veloso. Conservative q-improvement: Reinforcement learning for an interpretable decision-tree policy. *CoRR*, 2019.

[70] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 2019.

[71] Cynthia Rudin, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong. Interpretable machine learning: Fundamental principles and 10 grand challenges. *CoRR*, 2021.

[72] Cynthia Rudin and Berk Ustun. Optimized scoring systems: Toward trust in machine learning for healthcare and criminal justice. *INFORMS Journal on Applied Analytics*, 2018.

[73] Bernhard Schölkopf, Alex J. Smola, Robert C. Williamson, and Peter L. Bartlett. New Support Vector Algorithms. *Neural Computation*, 2000.

[74] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. USENIX Security '15.

[75] Andrew Silva, Matthew Gombolay, Taylor Killian, Ivan Jimenez, and Sung-Hyun Son. Optimization methods for interpretable differentiable decision trees applied to reinforcement learning. AISTATS '20.

[76] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. NeurIPS '12.

[77] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. NSDI '20.

[78] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. INFOCOM '16.

[79] Yingtian Tang, Han Lu, Xijun Li, Lei Chen, Mingxuan Yuan, and Jia Zeng. Learning-aided heuristics design for storage system. SIGMOD '21.

[80] Arnaud Van Looveren and Janis Klaise. Interpretable counterfactual explanations guided by prototypes. ECML-PKDD '21.

[81] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. OSDI '21.

[82] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. OSDI '20.

[83] Sarah Wiegreffe and Yuval Pinter. Attention is not not explanation. EMNLP '19.

[84] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. ICML '15.

[85] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. USENIX ATC '18.

[86] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. SoCC '17.

[87] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. SIGCOMM '15.

[88] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. USENIX ATC '21.

[89] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. SC '20.

[90] Ji Zhang, Ping Huang, Ke Zhou, Ming Xie, and Sebastian Schelter. Hddse: Enabling high-dimensional disk state embedding for generic failure detection system of heterogeneous disks in large data centers. USENIX ATC '20.

[91] Xinyang Zhang, Ningfei Wang, Hua Shen, Shouling Ji, Xiapu Luo, and Ting Wang. Interpretable deep learning under fire. USENIX Security '20.

[92] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. ASPLOS '21.

[93] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. OSDI '20.

[94] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. ASPLOS '20.

[95] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. Network planning with deep reinforcement learning. SIGCOMM '21.

[96] Kostas Zoumpatianos, Yin Lou, Ioana Ileana, Themis Palpanas, and Johannes Gehrke. Generating data series query workloads. *The VLDB Journal*, 2018.

A Supplementary Elaborations

In this section, we provide some supplementary elaborations about PRIMO algorithms, including algorithm detailed illustrations of PrAM and PrDT.

A.1 PrAM: Explainable Boosting Machine

Explainable Boosting Machine (EBM) [63] is a open-source implementation of Generalized Additive Models plus Interactions (GA^{2}M) [47] written in C++ and Python. Similar to popular GBDT algorithms (e.g. LightGBM [37]), the EBM training procedure begins by bucketing data from continuous features into discrete bins of histogram [62], which can significantly accelerate model training. Then EBM starts to learn shape function $f_i(\cdot)$ for each feature. Common choices for shape functions are regression splines, step functions and binary trees. For better prediction accuracy, it chooses the boosted trees, where each successive tree tries to predict the overall residual from all the preceding trees [46]. Furthermore, EBM optimizes the traditional boosting (greedy search) approach by leveraging *cyclic gradient boosting*, which learns a shallow tree for each feature in a round-robin fashion [62]. PrAM is heavily based on the implementation of EBM.

For simplicity and accuracy, PrAM leverages BO to automatically adjust model configurations, including the number of histogram bins, the number of considered interactions, learning rate, etc. It helps developers easily obtain the optimal model which is compact and accurate.

A.2 PrDT: Minimal Cost-Complexity Pruning

We adopt Minimal Cost-Complexity Pruning (CCP) [14] to prune the full tree in the *post-processing* stage. This algorithm aims to minimize a cost-complexity metric $R_\alpha(T)$ which is defined as

$$R_\alpha(T) = R(T) + \alpha \cdot N(T) \tag{3}$$

where $R(T)$ and $N(T)$ denote the misclassification cost (error rate) and complexity penalty (the number of leaves) of the decision tree T respectively. The trade-off between accuracy and sparsity of the tree is controlled by the *complexity parameter* α: as α increases, more leaves are pruned and the total impurity increases.

Figure 17 presents the impact of the *complexity parameter* α on the model performance and complexity for the LinnOS system (§4). When α is too small (before the red dashed line), PrDT has poor performance because of the severe overfitting. With a bigger α, developers could trade-off the model accuracy and complexity based on their system requirements. The optimal pruning factor can differ by many orders of magnitude for different systems according to our experiments. It is hard for system developers to identify an appropriate value of α intuitively. To address this, PRIMO adopts bayes optimization to perform post pruning automatically (§3.3.1).

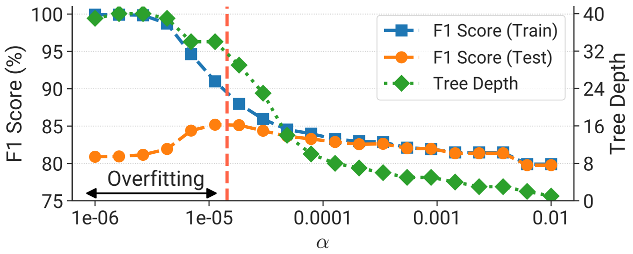


Figure 17: Performance (F1 Score) and complexity (Tree Depth) of the PrDT model under different α in LinnOS.

B Insufficiency of Existing Methods

To demonstrate the argument in §2.3, we provide some examples and perform analysis. These experiments reveal the insufficiency of both existing machine learning frameworks and interpretation tools.

B.1 Contradictory Interpretation

Clara adopts XGBoost to predict the optimal core counts for different NFs in Clara-MS. XGBoost contains a built-in api `get_score` to get the feature importance value of each feature and it can be regarded as the model interpretation. However, we argue that the interpretation has low fidelity. As evident from the Figure 18, interpretations based on different metrics present contrasting patterns, where both *gain* and *cover* are important intermediate metrics during model generation. For instance, R_{sum} is the second important feature according to *cover*-based explanation while seems ignorable from *gain*'s perspective. This makes developers feel confused which interpretation should trust. More seriously, this could mislead them to make wrong decisions, such as incorrectly omitting important input features while feature engineering.

B.2 Incapability for Global Surrogate

We have demonstrated the remarkable performance of PRIMO in this work. A common question is that *If using the existing interpretation tools for the model surrogate, can they also deliver equivalent effects?* Our answer is no. We evaluate two popular black-box interpretation methods: Lime [67] and Lemna [26]. Lime performs local interpretation through linear regression of data subset and Lemna adopts a mixture regression model to obtain interpretation in a similar way. Both of them are designed for local interpretation of several instances. However, we further explore whether they have the potential for the global model surrogate. Hence, we train the Lime and Lemna model on the LinnOS dataset, and compare the performance with PRIMO. As shown in Figure 19, Lime and Lemna cause much higher (7.3%~9.0%) false submit rates, which is the most significant metric for the system performance. As stated in §4, the overhead of failover (false revoke) is relatively negligible and all three methods perform well pertaining to this metric. Overall, existing interpretation tools are insufficient for the global model surrogate.

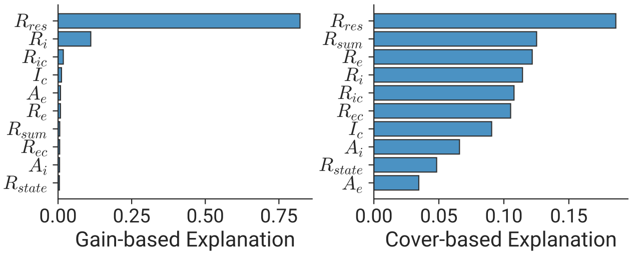


Figure 18: Interpretations of XGBoost model for Clara-MS task based on different metrics. Higher value indicates the feature is more important.

C Lessons Learned From the Case Studies

In order to draw high level conclusions from the three case studies discussed in §4, §5 and §6, we list the key observation of the PRIMO benefits for each system scenario as below. To summarize, in addition to model transparency, PRIMO provides higher prediction accuracy, smaller inference overhead and better model generalization ability. These greatly facilitate model deployment in practice.

1. LinnOS

Key Observation 1

The high inference overhead of the DNN model can hinder its deployment in practice, meanwhile seriously damaging the effect of system performance improvement. PRIMO successfully overcomes this bottleneck.

2. Clara

Key Observation 2

Instead of using various black-box models for different tasks, PRIMO uses a unified interpretable model achieving equivalent even better accuracy and endows capability of model adjustment to remedy the problem caused by drifted synthesis data.

3. Pensive

Key Observation 3

With PRIMO, we obtain an interpretable model that has better performance and generalization ability than the RL policy. Moreover, it achieves much less inference overhead and low distill cost for practical deployment.

D More Details about the Evaluated Systems

D.1 LinnOS

LinnOS infers the SSD speed using a lightweight neural network. The motivation behind LinnOS is that SSD read latency is unstable because some SSD internal operations (e.g., write-triggered garbage collection, buffer flushing) are contending with user read I/Os. In addition, there are the same replicas in other SSDs within the storage array (e.g., flash RAID) and we can utilize the built-in *failover* logic to circumvent slow read I/Os. The overhead of switching the *read* operation to a redundant SSD (namely *failover*) is ignorable compared with I/O pending time.

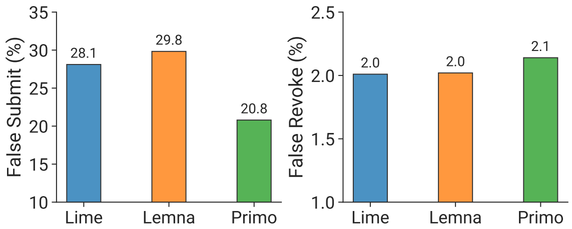


Figure 19: Comparison of Primo global surrogate performance with Lime and Lemna in the LinnOS scenario.

Implementation. We implemented PrDT models into the Linux kernel v5.4.8 (same version with LinnOS) within the *block* layer (written in C). We use the same SSD I/O traces in LinnOS, which were collected from Microsoft Bing Index servers and have been preprocessed by the authors.

Although the LinnOS workflow files are open-sourced on the Chameleon Cloud [38], the experiment results are unreliable due to the unstable and random SSD I/O accesses (also argued by the authors). Consequently, we shift our implementation and evaluation on a bare-metal server from CloudLab [18]. It contains four homogeneous enterprise-level 1.6TB SSDs. One of them serves as the system drive and the rest three SSDs are used for performance evaluation. Additionally, due to the rapid development of the SSD technology in recent years, the Microsoft traces collected in 2016 cannot give sufficient load pressure for evaluation. So we execute a constant writing task in the background for each SSD (LinnOS only records read I/Os). According to our numerous tests, the additional load will not cause fluctuations in the evaluation results.

For the PrDT interpretation results, PRIMO can provide a more precise visualized file of the PrDT which covers the number of samples at each flow and the Gini impurity value of each node rather than Figure 3.

D.2 Clara

Clara is a system that generates the optimal offloading strategies for NFs in SmartNIC. Since the performance characteristics of offloaded programs are opaque prior to porting and offloading strategies are difficult to reason about, developers need to first cross-port NFs to the SmartNIC, perform workload-specific benchmarks, and then iteratively tune the ported programs to achieve higher performance. Clara can analyze a legacy NF in its unported form and suggest porting strategies for the given NF to achieve higher performance.

Implementation. We follow the author-provided data preprocessing pipeline to deal with synthesized and real traces for training and evaluation. We conduct Clara evaluation on a

Notation	Description
A_i	IMEM Access
A_e	EMEM Access
I_c	Compute Intensity
R_i	IMEM / Overall Intensity Ratio
R_e	EMEM / Overall Intensity Ratio
R_{ic}	IMEM / Compute Ratio
R_{ec}	EMEM / Compute Ratio
R_{sum}	(IMEM + EMEM) / Compute Ratio
R_{res}	(IMEM − EMEM) / Compute Ratio
R_{state}	Non-Stateful / Stateful Compute Ratio

Table 5: Notation descriptions of Clara-MS.

Ubuntu 20.04 server with one Intel Core i9-10900 CPU, 64 GB memory and an NVIDIA RTX 2080Ti GPU. Note that because it is an offline system and we focus on model accuracy, Netronome SmartNIC is not required in our experiment.

Notation. We clarify the notations used in system interpretations (Figure 8). Table 5 shows processed features used in Clara-MS, where IMEM indicates SRAM-based internal memory and EMEM indicates DRAM-based external memory on SmartNICs. Instructions can be classified into Stateful (e.g., loads and stores to global variables in memory) and Non-Stateful (e.g., compute instructions, or accesses to function-local variables). Moreover, Overall Intensity represents the sum of IMEM, EMEM and Compute Intensity.

D.3 Pensieve

Pensieve [51] is a system that learns adaptive bitrate (ABR) algorithms automatically using RL technique. The online videos are stored on servers as multiple chunks (a few seconds of the video) and each chunk is encoded at several discrete bitrates (resolutions). Specifically, Pensieve adopts A3C [57] to perform RL training. Both the actor and critic networks use the same NN structure which contains a 1D-CNN layer and a fully connected layer. The actor takes recent network observations as input and suggests the bitrate for the next video chunk as the output. Content providers employ ABR algorithms to optimize user quality of experiment (QoE) which is defined as:

$$QoE = \sum_{n=1}^{N} q(R_n) - \mu \sum_{n=1}^{N} T_n - \sum_{n=1}^{N-1} |q(R_{n+1}) - q(R_n)| \tag{4}$$

where R_n represents the bitrate of the n^{th} chunk and $q(R_n)$ maps that bitrate to the quality perceived by a user. T_n represents the rebuffering time and the last term penalizes changes in video quality to favor smoothness.

Implementation. We employ the same server used in Clara for the Pensieve experiment. The video server is based on Apache httpd (Version 2.4.41) and uses Google Chrome (Version 96) as the client video player. We use 142 Norway HSDPA [68] network traces (provided by the authors) for evaluation, in addition, we process another 249 FCC [3, 54] traces (2018 version, follow the same preprocessing pipeline) for model generalization evaluation. Because the original FCC-18 network speed is much faster than HSDPA and we cannot distinguish the performance difference among algorithms, we scale down the network speed by 4 times to keep consistent with FCC-16 with regards to the median values.

Notation	Description
X_t	Past chunk throughput
T_t	Past chunk download time
N_k	Next chunk sizes
B	Current buffer size
C	Number of chunks left
L	Last chunk bitrate

Table 6: Notation descriptions of Pensieve.

We obtain `PrDT` model through distilling from the trained RL actor model (pre-trained model that Pensieve authors provided). After that, we implement `PrDT` (written in JavaScript) into *ABRController* of dash.js [2] (Version 2.4). For the test videos, we use the same video in Pensieve at bitrates in {300, 750, 1200, 1850, 2850, 4300} kbps (which pertain to video modes in {240, 360, 480, 720, 1080, 1440}p). This video is divided into 48 chunks and each chunk represented approximately 4 seconds. Furthermore, we adopt $q(R_n) = R_n$ in Equation 4 to set *linear QoE* as the our evaluation metric.

Notations. We clarify notations used in system interpretations (Figure 12). Table 6 shows environment state variables used in Pensieve to generate bitrate decision, where X_t and T_t denotes past sequences of throughput and download time respectively ($t = 1, \cdots, 8$). Moreover, N_k represents sequence of next chunk sizes ($k = 1, \cdots, 6$).

E Artifact

Primo and case study scripts are available as below. To reproduce the main results of this work, we also provide detailed documentation and instructions in the artifact repository.

Artifact Links

GitHub: https://github.com/S-Lab-System-Group/Primo
DOI: https://doi.org/10.5281/zenodo.6529892