

Prediction-Guided Control in Data Center Networks

Kevin Zhao¹, Chenning Li², Anton A. Zabreyko², Arash Nasr-Esfahany², Anna Goncharenko¹, David Dai¹, Sidharth Lakshmanan¹, Claire Li¹, Mohammad Alizadeh², Thomas E. Anderson¹

¹University of Washington, ²MIT CSAIL

Abstract

In this paper, we design, implement, and evaluate Polyphony, a system to give network operators a new way to control and reduce the frequency of poor tail latency events in multi-class data center networks, on the time scale of minutes. Polyphony is designed to be complementary to other adaptive mechanisms like congestion control and traffic engineering, but targets different aspects of network operation that have previously been considered static. By contrast to Polyphony, prior model-free optimization methods work best when there are only a few relevant degrees of freedom and where workloads and measurements are stable, assumptions not present in modern data center networks.

Polyphony develops novel methods for measuring, predicting, and controlling network quality of service metrics for a dynamically changing workload. First, we monitor and aggregate workloads on a network-wide basis; we use the result as input to an approximate counterfactual prediction engine that estimates the effect of potential network configuration changes on network quality of service; we apply the best candidate and repeat in a closed-loop manner aimed at rapidly and stably converging to a configuration that meets operator goals. Using CloudLab on a simple topology, we observe that Polyphony converges to tight SLOs within ten minutes, and re-stabilizes after large workload shifts within fifteen minutes, while the prior state of the art fails to adapt.

1 Introduction

Improving network quality of service – the likelihood that network transfers complete in a timely fashion – is a long-standing goal of networking research [10, 12, 14, 15, 46]. Data center applications in particular are often very latency sensitive [8, 20, 35, 56], with inherent dynamic variability [40], a predominance of short flows [24], and frequent incast and outcast traffic patterns [47]. Link oversubscription to reduce costs, as well as link and switch failures add to the challenges. A number of techniques have been proposed, including resource reservations [12], priority scheduling [10], pricing signals to discourage usage during periods of contention [15], receiver-based management of incast communication [26, 37], switch behavior to bypass queues [2, 5, 24, 25, 45, 49], and central coordination of when hosts are allowed to transmit [28, 39].

In practice, most production data center networks combine three ideas to improve network quality of service, albeit with mixed results. First, aggressive congestion control algorithms are designed to react quickly (within a few round trips) and forcefully to observed congestion along a path in an attempt to

keep queues small [4, 5, 30, 33]. Second, on longer time scales, traffic engineering (such as weighted ECMP and Optical Circuit Switching (OCS)) attempts to balance traffic volumes over available paths despite non-uniform topologies and inherent temporal and spatial variability in traffic demand [40]. Third, as a final backstop, mission critical or latency sensitive traffic is assigned to separate traffic classes to isolate it from other competing traffic, typically using scheduling weights rather than priorities to avoid starvation for low priority traffic. Even with these mechanisms in place, well-managed data center networks still experience significant tail slowdowns for RDMA and RPC operations caused by network congestion [8, 44].

In this paper, we design, implement, and evaluate Polyphony, a system to give network operators a new way to control and reduce the frequency of poor tail latency events in multi-class data center networks. Polyphony is designed to be orthogonal and complementary to other workload-adaptive mechanisms like congestion control and traffic engineering. We assume the network operator sets some traffic class specific performance goal, such as keeping the 99th percentile flow completion time (FCT) slowdown – defined as FCT normalized to its lowest value in an unloaded network – below a threshold.

What additional mechanisms do operators have to effectively control the FCT? Modern data center networks expose a broad control surface, ranging from various end-host congestion control parameters to switch-level queueing, scheduling, and buffer allocation policies. Today, these parameters are typically configured once, or adjusted only in response to emergencies. However, we show that the effects on tail latency can be profound: small changes in end host or switch behavior can reshape latency distributions to achieve, or fail to achieve, operator goals. Further, the appropriate changes to make are highly workload dependent and therefore time varying – burstiness, correlated behavior, and even the behavior of other traffic classes can affect user-observable outcomes, in ways that are hard to define from first principles.

Our approach is to develop a new *prediction-guided controller* for data center networks that rapidly converges, in response to workload or topology changes, over a period of minutes. Unlike purely reactive congestion control that collects evidence on a single path at a time, Polyphony samples and aggregates network workload data on a network-wide basis to reduce the likelihood of future congestion events *before* they happen. Unlike traffic engineering, Polyphony samples much richer workload information than just traffic volumes.

We leverage recent advances in fast approximate models for estimating network performance. These models predict tail

latency for a given network workload, topology, and configuration [21, 32, 53–55]. They can produce approximate answers for how network performance may change as a result of some control action, many orders of magnitude faster than detailed packet-level simulation frameworks like ns-3 [38]. Specifically, we use two open source models, Parsimon [55] and m3 [32]. We extend them to support class-based queueing and to work on live network traffic, specifically CloudLab [16]. Because m3 uses machine learning, we train it using CloudLab measurements as ground truth; the original work trained against ns-3. CloudLab introduces various sources of variability not present in ns-3, such as host jitter that measurably affects how the network is used. As a result, our CloudLab-trained m3 is substantially more accurate than both the original m3 and Parsimon. This allows us to evaluate robustness to prediction error. We find that a slower and less accurate predictor slows convergence. We show this sensitivity in our experiments, using m3 as the higher-fidelity predictor and Parsimon as a coarser baseline.

The general approach of prediction-guided network control poses several problems which this work addresses. First, network prediction models have significant error, and we show that this means they cannot be used directly to recommend configurations (or if we do, then we end up stuck at a non-optimal operating point). Rather, as we explore the configuration space, we train a Gaussian model to correct for observed model error between the predicted and live system. Since this corrected model is most accurate in the neighborhood of previous trials, we bias the search to favor exploration within a trust region around the best known configuration. Finally, we use a denoiser to smooth the results across different trials to reduce sensitivity to measurement variance. We show through an ablation study that each of these steps is necessary to efficiently reach a good operating point.

As a proof of concept, we evaluate Polyphony using each of the two prediction models on CloudLab for a small network, and the more accurate m3 model on ns-3 to study behavior on larger-scale networks, varying nine network configuration parameters. Relative to SelfTune, a state of the art model-free optimizer, given tight class-specific tail latency goals and starting from a non-optimal configuration, Polyphony using m3 converges on CloudLab to a configuration that meets the objective within ten minutes of real time, while SelfTune does not make much progress even within the sixty-minute trial period. When we introduce large periodic workload changes, SelfTune shows no clear signs of adaptation, while Polyphony re-stabilizes to SLOs within fifteen minutes. Polyphony’s source code will be publicly available.

2 Background and Motivation

The setting for our work is a network where traffic is divided into multiple service classes, each with its own tail latency objective. For example, an operator might require that 99% of flows in one class must have a flow completion time (FCT)

slowdown no greater than 10×, while another class may be able to tolerate a slowdown up to 20×. Following Mogul and Wilkes [36], we call this target a service level objective (SLO), and the measured performance metric (P99 FCT slowdown) a service level indicator (SLI). The bound (e.g., 10 or 20×) is the SLO threshold.

Our approach is agnostic to the unit of measurement, but to be concrete we assume it is a remote procedure call (RPC), remote memory operation (RDMA), or independent data transfer. Latency is measured as the time to complete the transfer, including the transmission, propagation, and queueing delay, from when the first packet is available to be sent until the last packet arrives at the destination. We include in the latency any queueing at the end host needed for traffic shaping or congestion control. Following industry practice, we modify the host operating system to sample tail latencies as input to our control system.

We define FCT slowdown as the flow latency divided by the minimum latency on an unloaded network. For example, in a network with a round trip propagation delay of 10 μsec and 10 Gbps links, the minimum latency for a 12.5KB transfer would be 20μsec. Polyphony supports performance targets to be defined separately for different message sizes, e.g., to ensure that medium-sized messages aren’t starved. In this paper, we focus on aggregate tail performance.

We assume all network hosts and switches are configured identically, but in a class-aware manner. Each traffic class is assigned its own dedicated queue at each network link; each class is given a scheduling weight that controls how frequently packets from that class are scheduled when other traffic is present. All hosts and switches implement the same congestion control algorithm, but the parameters controlling the behavior of the algorithm, such as the initial window size or its overall aggressiveness, are traffic class specific. The prediction models we use generalize across many commonly used congestion control algorithms [32, 55], but for our proof of concept we narrow our focus to DCTCP [4]. It contains an initial congestion window and a threshold parameter, K , for switch queues to trigger end hosts to reduce their sending rate; these parameters are allowed to vary by traffic class.

Whether a specific traffic class meets its objective depends on the specific workload and parameters of that class, but also that of other classes. Class-based switch scheduling allocates a worst case minimum share of the line rate to each class in proportion to its weight, but scheduling is also work conserving. If a class does not have traffic present (e.g., because it is bursty) the unused capacity is proportionately split among the other traffic classes with traffic present. As a result, the effect of individual changes may not be monotonic. For example, the DCTCP K value controls the target amount of queueing at congested switches. Reducing this value can improve tail latency for flows that fit within the initial congestion window, but it can also hurt tail latency by reducing the ability of the traffic class to take advantage of the idle periods of other classes.

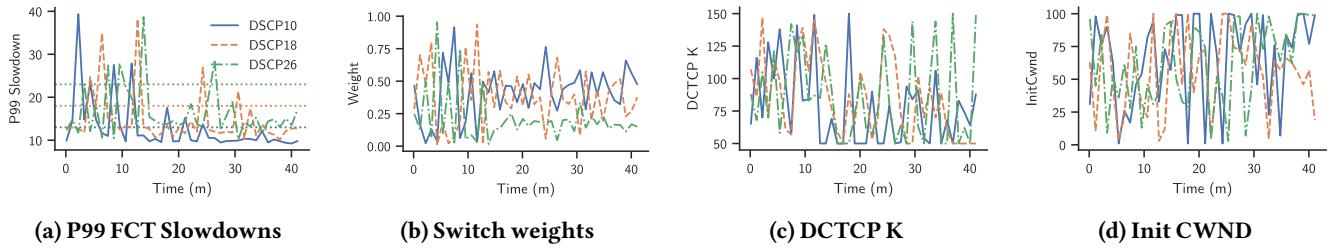


Figure 1: Model-free Bayesian optimization leads to poor behavior during convergence. Figure 1a shows the P99 FCT slowdown of three traffic classes over time. Horizontal lines indicate SLO thresholds. Both configurations and tail FCT slowdowns oscillate as the optimizer explores the parameter space.

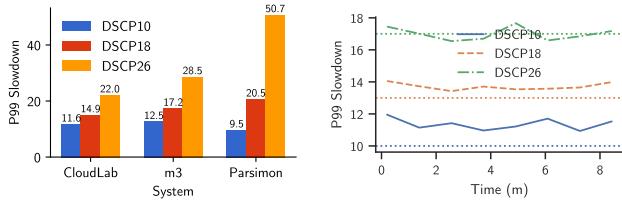


Figure 2: Model error and its consequences for three classes with varying SLOs. m3 is trained on CloudLab while Parsimon uses packet-level simulation. (a) Even m3 mispredicts CloudLab’s P99 FCT slowdown by 18% on average. (b) When the best configuration predicted by m3 is applied to the network, all three traffic classes violate their SLOs, even though they can all be met simultaneously. The configuration is kept the same across samples; variance is due to noise.

2.1 Existing methods

Given that the network exposes parameters for control, our goal is not a “best configuration” but continuous regulation: adjusting settings in closed loop to track SLOs as workloads evolve. When operating conditions change—for example a shift in traffic patterns or workload intensity—the controller should adapt while minimizing SLO violations. We therefore evaluate approaches by: (i) convergence time (time to reach per-class SLO compliance), (ii) regret (cumulative deviation from compliance during adaptation), and (iii) fairness (whether deviations are borne evenly across classes). In our experiments, we treat the SLOs for each traffic classes as equally important, but Polyphony can also prioritize certain classes with weighted objectives.

Model-free autotuners. A prominent line of prior work adjusts parameters using black-box autotuning. These methods are model-free: they iteratively deploy settings and observe outcomes on the live system. For stability, they typically operate on long time horizons, with each iteration lasting up to hours or days. A common example is Bayesian optimization [13], which selects configurations to test by balancing exploration (trying uncertain settings) and exploitation (choosing promising ones). While model-free methods can eventually find good configurations, they must search by running real experiments on the target system, risking SLO violations (regret) in the process. Figure 1 shows standard

Bayesian optimization tuning a small CloudLab network with three traffic classes and nine parameters (per-class switch weight, DCTCP marking threshold, and initial congestion window). Although the optimization eventually settles after about thirty minutes, it first explores widely, causing users of the system to experience significant SLO violations. We would expect these oscillations to be more pronounced with more traffic classes and additional parameters.

An emerging class of model-free autotuners uses reinforcement learning (RL). RL tuners like SelfTune [29] and OPPERTUNE [48] update parameters by acting in discrete rounds: in each round, the tuner picks a configuration, the system runs for some time—typically ranging from an hour to days—and a scalar reward is fed back. These approaches rely on round-level measurements being information-rich and averaging out noise. In our setting with short timescales and noisy tail latency measurements, these conditions do not hold. As a result, reward signals can be noisy and gradients weak, causing updates to shrink and learning to stall. We apply SelfTune to our setting in the evaluation (Section 8). We find that, whether by Bayesian optimization or reinforcement learning, model-free autotuning struggles to track tail latency SLOs on short timescales in dynamic, noisy environments.

Emerging fast models. Meanwhile, recent projects like Parsimon and m3 have developed fast, approximate simulation methods for predicting tail latency in data center networks. One idea would be to use their predictions to drive configurations directly in open loop. Unfortunately, as Figure 2a shows, these models still have significant error. Even m3, which can be trained directly on the target platform, exhibits in this scenario an average of about 18% error in predicting 99th percentile FCT slowdown across traffic classes. Figure 2b shows what happens when we use the model to find its predicted best configuration, and then apply it directly. Even though the SLOs are achievable, the deployed configuration misses all SLOs.

Model-free search learns from live measurements but can incur SLO violations due to wide exploration; model-based optimization can avoid unsafe exploration but inherits the model’s errors. Neither alone provides fast, low-regret adaptation for tail latency SLOs. Polyphony combines aspects of both

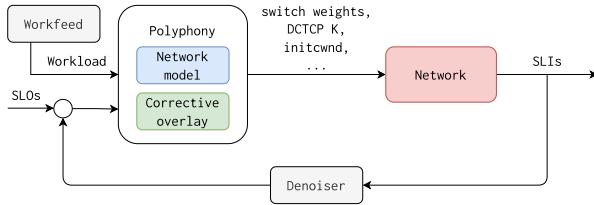


Figure 3: Polyphony’s closed-loop architecture.

approaches by coupling a fast model with online corrective feedback. The next section outlines Polyphony’s closed-loop architecture and introduces how it blends simulation and live measurement to achieve SLO compliance.

3 Polyphony Overview

Given a live network that serves multiple traffic classes with per-class tail latency SLOs, our goal is to continuously adjust a set of knobs—like switch weights or congestion control parameters—so all SLOs are met and tail latency is minimized. As a secondary goal, allocations should be *fair* in the sense that no class is persistently disadvantaged relative to its SLO. We frame this as a regulation problem: Polyphony acts as an online controller that tracks per-class SLO constraints via small, bounded adjustments.

Figure 3 depicts Polyphony’s closed-loop architecture. The core idea is to combine a cheap, approximate performance model with an online-learned corrective overlay. Polyphony then uses the overlay to compute the next bounded adjustment within a trust region for a network setting to try next. By optimizing a predictive model rather than the live system, Polyphony reduces the number of costly system measurements required to find good configurations, and limits the risk of applying undesirable settings. The system takes as input the target SLOs and the current service-level indicators (SLIs), and it outputs the next setting to apply. The performance models require workload information as input. To measure workloads, Polyphony bundles a lightweight Workfeed component that uses eBPF to capture flow events and reconstruct flow sequences for the models.

At each control interval, Polyphony computes a bounded update to the control knobs. First, it collects SLIs, including per-class tail slowdowns. These metrics capture the performance seen by each class and serve as feedback for updating the corrective overlay. Next, it filters these measurements to reduce the effects of noise and transient spikes, producing a smoothed estimate of the system performance. Using this filtered signal, Polyphony updates its corrective overlay. This model captures the discrepancy (i.e., residual) between the predictions of the fast, approximate simulator and the live measurements from the real system. We use the estimated residual to improve the predictive accuracy of the overall model. Then, Polyphony searches for a set of network knobs within a trust region that maximize the expected performance

improvement. This step balances exploration with caution, ensuring that new settings are only tried if the model predicts an improvement with high confidence. Finally, the selected configuration is sent to the switches and hosts to actuate the intervention.

The following sections describe Polyphony’s methods in detail. First, we focus on settings with stable workloads in Section 4. Then, we describe adaptation to workload changes in Section 5.

4 Safe Prediction-Guided Optimization

To start, we view the network as a black box that, for a workload, accepts a vector of parameters \mathbf{x} —switch weights, marking thresholds, etc.—and emits a vector of SLIs \mathbf{y} , such as per-class tail latencies. Throughout this section, we assume the workload is fixed and stable, and drop it from notations to avoid clutter. Alongside the network, we have an approximate model, like Parsimon [55] or m3 [32], that takes the same \mathbf{x} and produces *predicted* SLIs for an independent sample of the same workload. We assume that the model makes errors but is cheap to evaluate, and that the real system is noisy and risky to probe. Polyphony’s principal idea is to learn the difference between them and then optimize the performance using a *corrected* version of the model. It also optimizes within a safety envelope to avoid straying from known good configurations. To make these ideas more precise, we begin by introducing some notation.

4.1 Definitions and Problem Formulation

We first define the parameter space. Polyphony considers two kinds of parameters. The first lie in a hypercube, where each parameter varies independently. Examples of these are congestion control parameters or initial congestion windows. The second lie in a simplex: they are constrained to be non-negative and sum to one. These represent allocations or proportions, such as switch weights which must be distributed among traffic classes. Polyphony manipulates a vector

$$\mathbf{x} = (h^{(1)}, h^{(2)}, \dots, h^{(M_h)}, u^{(1)}, u^{(2)}, \dots, u^{(M_u)}),$$

where $h^{(k)} \in [0,1]^{d_k}$ are hypercube parameters and $u^{(\ell)} \in \Delta^{d_\ell}$ are simplex parameters. Combining these domains, the full parameter space is

$$X = \left(\prod_k [0,1]^{d_k} \right) \times \left(\prod_\ell \Delta^{d_\ell} \right).$$

All coordinates are scaled to $[0,1]$ for optimization and inverse-normalized for actuation.

For the particular workload, we think of the system as a function $s(\mathbf{x}) : \mathcal{X} \rightarrow \mathbb{R}^N$ which maps a vector of parameters to a vector of SLIs. The model $m(\mathbf{x}) : \mathcal{X} \rightarrow \mathbb{R}^N$ is defined similarly for the same workload. Although $s(\mathbf{x})$ and $m(\mathbf{x})$ each return a N -dimensional vector of SLIs, an optimizer needs a total order over configurations to decide whether one setting is better than another. Polyphony therefore collapses the vector

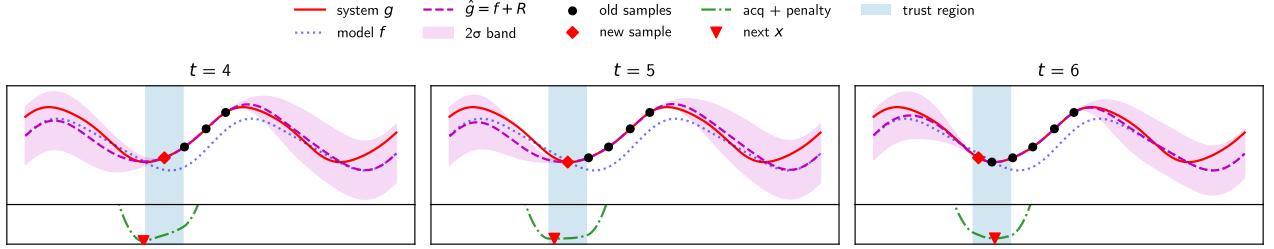


Figure 4: An illustration of Bayesian optimization with a residual surrogate and a trust region. Each panel shows one control iteration ($t = 4, 5, 6$). The acquisition function (bottom curve) is quadratically penalized outside the trust region. Over successive steps the surrogate improves near the new sample and the trust region recenters, guiding the optimizer toward better configurations while avoiding unsafe exploration.

into a single scalar cost J . Doing so has two main advantages: 1) it reduces the problem to scalar optimization with a clear improvement signal, and 2) it allows us to express SLO compliance and fairness as a single quantity to optimize instead of exploring a Pareto frontier.

To guide optimization, we need a scalar objective that reflects our goals of SLO compliance and fairness. Let $\mathbf{y} = (y_1, \dots, y_N) \in \mathbb{R}^N$ be the observed SLIs (e.g., tail latencies) for each of the N traffic classes, and let $\mathbf{v} = (v_1, \dots, v_N) \in \mathbb{R}^N$ be their corresponding SLOs thresholds. To ensure SLO compliance, we want each class’s observed slowdown y_i to be less than its threshold v_i , so we define the per-class ratio $r_i = y_i/v_i$. A value $r_i \leq 1$ indicates compliance. Optimizing for SLO compliance then becomes minimizing some aggregate of these ratios across classes.

We choose the maximum as our aggregate because SLO violations are typically unacceptable even if they affect only a single class. Using the maximum of the ratios prioritizes reducing the worst violation. However, maximum is non-differentiable, which prevents the use of gradient-based optimization methods. To address this, we use LogSumExp (LSE) [11, Sec. 3.1.5] as a smooth approximation of maximum, with sharpness controlled by parameter $c > 0$:

$$\text{LSE}(r_1, \dots, r_N) = \frac{1}{c} \ln \left(\sum_{i=1}^N \exp(cr_i) \right).$$

As $c \rightarrow \infty$, the function approaches a sharp maximum, while smaller values of c yield a smoother behavior.

To encourage fair allocations, we add a fairness penalty which tries to equalize the r_i ratios. Let $\bar{r} = \frac{1}{N} \sum_{i=1}^N r_i$ be the mean of the ratios. We define

$$\text{fairness}(r_1, \dots, r_N) = \frac{1}{N} \sum_{i=1}^N |r_i - \bar{r}|.$$

The final objective is then

$$J(r_1, \dots, r_N) = \text{LSE}(r_1, \dots, r_N) + \lambda \text{fairness}(r_1, \dots, r_N),$$

where $\lambda \geq 0$ is a trade-off constant. Lower values of J indicate a smaller worst-case slowdown ratio and/or a fairer spread of ratios across classes, depending on the weight λ .

Recall that for a particular workload, the system $s(\mathbf{x})$ and model $m(\mathbf{x})$ each return a vector of SLIs given a configuration \mathbf{x} . To calculate r_i ratios used for objective (J) computation, we can use $s(\mathbf{x})$ or $m(\mathbf{x})$. We call the former $g(\mathbf{x})$ and the latter $f(\mathbf{x})$. In words, $g(\mathbf{x})$ is the system’s observed performance and $f(\mathbf{x})$ is the model’s predicted performance. In what follows, we refer to g as “the system” and f as “the model.”

4.2 Modeling Residuals

The controller must minimize a system g that is noisy and risky to sample while taking advantage of a model f that could have significant modeling error. Directly learning $g(\mathbf{x})$ is difficult because it can be highly nonlinear and discontinuous. Instead, Polyphony learns the *residual*

$$R(\mathbf{x}) = g(\mathbf{x}) - f(\mathbf{x}),$$

which is typically smoother and smaller in magnitude.

Gaussian-process surrogate. We treat the residual as a random function drawn from a Gaussian process $R \sim \text{GP}$. Given a set of residual observations $\{(\mathbf{x}_j, R_j)\}$, standard GP inference returns a posterior mean $\mu_R(\mathbf{x})$ and variance $\sigma_R^2(\mathbf{x})$. We use $\hat{g}(\mathbf{x}) = f(\mathbf{x}) + \mu_R(\mathbf{x})$ as the bias-corrected predictor and $\sigma_{\hat{g}}^2(\mathbf{x}) = \sigma_R^2(\mathbf{x})$ as its uncertainty. This uncertainty estimate provides the risk budget that supports the safe optimization strategy that follows.

4.3 Bayesian Optimization over \hat{g}

Our surrogate \hat{g} supplies both predictions and calibrated uncertainty. This leads naturally to the use of Bayesian optimization to balance exploration and exploitation.

Bayesian optimization (BO) iteratively chooses the next configuration by maximizing an *acquisition function* built on top of a fast *surrogate* of the true objective. In our case

$$\text{surrogate} = \hat{g}(\mathbf{x}) \quad \text{from §4.2},$$

and the acquisition is Expected Improvement (EI).

Expected Improvement. Let $g_{\text{best}} = \min_{j < t} g(\mathbf{x}_j)$ be the smallest *observed* objective after $t-1$ iterations. For a candidate \mathbf{x} with surrogate mean μ_g and standard deviation σ_g ,

$$\text{EI}(\mathbf{x}) = (g_{\text{best}} - \mu_g) \Phi(z) + \sigma_g \phi(z), \quad z = \frac{g_{\text{best}} - \mu_g}{\sigma_g},$$

where $\Phi(\cdot)$ and $\phi(\cdot)$ are the standard normal CDF and PDF. EI is the *expected drop in the objective* if we were to sample at \mathbf{x} . Anchoring EI to the smallest *measured* value guards against model over-optimism: a point can only show positive improvement if it is expected to beat what the system has actually achieved, not merely what the model predicts. Because Polyphony is a minimizer, the implementation minimizes $A(\mathbf{x}) = -\text{EI}(\mathbf{x})$:

$$\mathbf{x}_t = \operatorname{argmin}_{\mathbf{x} \in \mathcal{X}} A(\mathbf{x}).$$

EI balances exploitation of low μ_g with exploration in regions of high σ_g , aiming for sample-efficient improvement over time.

Handling simplexes. Bayesian optimization typically operates in an unconstrained, Euclidean space. To accommodate simplex-valued parameters—such as traffic class weights that must be non-negative and sum to one—we optimize in the unconstrained pre-image and map the result through a softmax transformation. This allows gradient-based methods to search freely while ensuring that the final parameter vector lies in the simplex. The surrogate and acquisition functions are defined over the unconstrained space but always evaluated on the mapped simplex point.

4.4 Safe Exploration

While Bayesian optimization efficiently explores the parameter space \mathcal{X} by balancing exploration and exploitation, it does not guarantee safety. Probing g can be risky: unconstrained exploration might lead the optimizer to query points \mathbf{x} that violate SLOs, even if the surrogate \hat{g} predicts otherwise.

Trust region. To mitigate this risk, Polyphony uses a safe exploration strategy that builds on well-understood methods from the optimization literature [17]. Instead of optimizing the acquisition over the entire parameter space \mathcal{X} , we restrict the search to a *trust region* centered around the *best configuration* observed so far. Let $\mathbf{x}_{\text{best}}^{(t)} = \operatorname{argmin}_{\mathbf{x}, j < t} g(\mathbf{x}_j)$ be the parameters corresponding to the best *observed* system performance $g_{\text{best}} = g(\mathbf{x}_{\text{best}}^{(t)})$ after $t-1$ iterations. The optimizer at step t focuses its search within a neighborhood of $\mathbf{x}_{\text{best}}^{(t)}$.

Distance metric. Defining this neighborhood requires a distance metric on the mixed parameter space \mathcal{X} . Each configuration $\mathbf{x} \in \mathcal{X}$ consists of 1) a hypercube part $\mathbf{h} \in \prod_{k=1}^{M_h} [0, 1]^{d_k}$, and 2) M_u simplex blocks $u^{(1)}, \dots, u^{(M_u)} \in \Delta^{d_t}$. We measure separation with

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\underbrace{\|\mathbf{h}_x - \mathbf{h}_y\|_2^2}_{\text{hypercube}} + \alpha \sum_{\ell=1}^{M_u} \underbrace{d_A(u_x^{(\ell)}, u_y^{(\ell)})^2}_{\text{simplex}}},$$

where d_A is the Aitchison distance commonly used for compositional data [1] and $\alpha \geq 0$ balances the Euclidean and compositional parts. The weight α is tuned so that the simplex and hypercube terms contribute on the same numerical scale.

Enforcement. Enforcing a hard constraint $d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) \leq \epsilon$ during optimization can be complex, especially for simplex parameters, which we optimize in an unconstrained space. Polyphony enforces the trust region by adding a penalty term to the acquisition function:

$$\text{penalty}(\mathbf{x}; \mathbf{x}_{\text{best}}^{(t)}, \epsilon) = \begin{cases} 0 & d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) \leq \epsilon, \\ \beta \cdot (d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) - \epsilon)^2 & \text{otherwise,} \end{cases}$$

where ϵ is the current trust-region radius and $\beta > 0$ controls the penalty severity. The optimizer minimizes

$$A_{\text{TR}}(\mathbf{x}) = -\text{EI}(\mathbf{x}) + \text{penalty}(\mathbf{x}; \mathbf{x}_{\text{best}}^{(t)}, \epsilon)$$

thereby searching freely *inside* the ball $d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) \leq \epsilon$ while rapidly discouraging excursions beyond it. Because the penalty is quadratic and continuous, gradients remain well-behaved. In our experiments, we set $\beta = 10^4$.

Figure 4 illustrates the concepts we have introduced so far. Next, we describe how Polyphony adapts to new operating regimes, and how it handles data to remain both reactive and stable over long time horizons.

5 Adaptation and Data Quality

Thus far, we have described methods that allow Polyphony to converge to good solutions so long as g remains stationary. Real networks, however, can experience step changes: traffic mixes shift, hot spots appear. To adapt to new regimes, Polyphony must 1) detect that the regime has changed, and 2) forget stale data that would mislead the GP. This section discusses these mechanisms. In addition, we describe methods to ensure Polyphony remains effective over long timescales.

5.1 Adapting to changing g

Consider how Polyphony—as currently described—would respond if the network suddenly experienced a disruption, such as a sudden influx of traffic. The value of our objective g would spike, prompting the controller to adjust parameters to achieve SLO conformance. However, the trust region would still be centered around the old best point, tying the optimization to that region and causing it not to make progress. To prevent this, we first maintain a *rolling* best objective rather than a static one:

$$g_{\text{best}}(t) = \min_{j=t-W_g}^{t-1} g(\mathbf{x}_j),$$

where W_g is the size of the rolling window. Next, we implement a simple scheme for objective spike detection and trust region reset. If the current measurement exceeds the rolling best objective by a factor of ϕ for k consecutive iterations,

$g(\mathbf{x}_t) > \phi g_{\text{best}}(t)$, we treat the event as a regime shift, and we 1) reset the g_{best} window and 2) designate the current configuration as $\mathbf{x}_{\text{best}}^{(t)}$.

5.2 Processing data samples

Rolling data. As Polyphony runs over time, the number of samples it collects can grow without bound. This behavior is undesirable for two reasons. First, GP inference scales as $O(n^3)$, where n is the number of samples. Second, old points from outdated regimes can dominate, forcing the surrogate to confidently explain behavior that is no longer relevant, making adaptation sluggish. To eliminate this effect, we maintain a rolling window of samples of size W_s .

Filtering data. As Polyphony converges to a good solution, every new sample is likely to lie in the same small neighborhood, and thus nearly identical in the input space. Updating a GP requires inverting a covariance matrix, which can become ill-conditioned if most of its rows are almost the same. This can cause the GP posterior mean and variance to become unreliable. Polyphony employs a simple data admission policy to mitigate this risk. We say a configuration is *novel* if it is far enough away from all other configurations in the rolling window of samples, and we say a sample is *surprising* if the GP mispredicts its outcome. Polyphony admits a sample if at least one of these conditions is met. This rule acts as a lightweight guard that keeps the GP numerically stable.

Denoising data. Real systems are noisy, especially when measuring metrics like tail latency. Feeding jittery data into the GP can lead it to interpret noise as signal, slowing learning, and chasing spurious spikes. Before updating the GP with a new sample, Polyphony applies a causal low-pass Gaussian smoother whose width adapts to the volatility of the signal. If the signal becomes completely stable, the filter automatically turns off. We evaluate the effect of the denoiser in Section 8.4.

5.3 Measuring workloads online

Recall that the performance models Polyphony uses require workload estimates—representative sequences of flows with their sizes, classes, and arrival times—as input. In a live network, obtaining these estimates requires capturing flow-level metrics from production traffic. We built **Workfeed**, a light-weight distributed system that monitors ongoing traffic and provides workload traces for model evaluation. Workfeed measures TCP flows to mirror the workload generator; extending it to capture RPC-level traces over long-lived connections is future work.

Flow-level monitoring. Workfeed uses eBPF probes attached to TCP socket lifecycle events (`inet_sock_set_state`, `tcp_destroy_sock`) to capture flow metadata. For each completed flow, the system records the 5-tuple, DSCP marking

(for traffic class discrimination), bytes transferred, and timestamps. The probes operate with minimal overhead and use a zero-copy ring buffer design for efficiency.

Rack-level aggregation. Flow records are collected from individual hosts and aggregated at the rack level. Each rack runs a aggregator process that receives flow records via UDP and forwards them to the controller. Optionally, the aggregator can downsample flows using deterministic hash-based sampling and assign weights for statistical reconstruction, though in our evaluation, we capture complete flow traces and replay them exactly.

Performance impact. Workfeed’s kernel-level monitoring introduces negligible overhead. Each flow record is compact (48 bytes), and the system is designed to support tracking tens of thousands of concurrent flows per host. In our dynamic adaptation experiments, Workfeed successfully captured workloads without measurable impact on application latency or throughput.

6 Limitations

This section discusses Polyphony’s limitations.

No formal guarantees. Polyphony provides no formal guarantees of convergence to globally optimal configurations. Convergence depends on model quality and the effectiveness of the sampling strategy. The trust region ensures monotone improvement in expectation inside one regime, but global optimality is not guaranteed. This safety-performance tradeoff is typical of local optimization methods, which prioritize safe, incremental improvements over global exploration.

Scaling with dimensionality. As the number of adjusted knobs increases, so does the sample complexity required for accurate modeling and optimization. While Polyphony uses Gaussian Process mixtures [31] to scale to moderate dimensions, they remain inherently limited by the curse of dimensionality. This limits Polyphony’s scalability in very high-dimensional control problems without additional techniques like dimensionality reduction or structured modeling.

No categorical knobs. Polyphony currently only handles continuous parameters; categorical choices are not supported. A straightforward extension is to one-hot encode each category and treat the resulting binary dimensions as additional continuous inputs. We leave this to future work.

Controller hyperparameters. As currently formulated, Polyphony introduces a set of hyperparameters, including data window size, trust region size, regime shift detector thresholds, and others. All experiments in this paper use a single, fixed setting for each. In practice, we have found that most parameters could be set once and did not require tuning. Some, like the trust region radius ϵ , data window size W_s , and regime shift thresholds ϕ and k may need to be adjusted depending on model quality and traffic volatility.

7 Implementation

We implement Polyphony as a modular controller framework in Rust encompassing workload generation, performance modeling, runtime correction, optimization, and system actuation.

Workload generation. We develop `emu`, a workload generator that emits TCP flows using configurable workloads (e.g., flow size and inter-arrival distributions). Traffic classes are marked with DSCP values (10, 18, 26), enabling per-class policy control. `emu` uses Prometheus [41] to collect aggregate flow metrics (e.g., p99 FCT slowdown) to feed to the controller.

Performance models. We extend both Parsimon [55] and m3 [32] to operate over a 9-dimensional parameter space consisting of switch weights, initial congestion windows, and DCTCP marking thresholds for three traffic classes. m3 is a machine learning-based predictor trained on data collected from the target deployment environment (CloudLab or ns-3, depending on the experiment) using Facebook’s public traces [43]. For Gaussian processes in the corrective overlay, we use the `egobox-moe` [31] library, automatically selecting the best-fit kernel and mean function using the Bayesian Information Criterion.

System actuation. For experimentation in CloudLab [16], we implement actuators for per-class switch weights, DCTCP marking thresholds, and initial congestion windows. The first two are configured via the DellS4048 switch’s CLI; the last uses the `ip route` utility. We also add support for tuning these parameters in ns-3 [38] by adapting the HPCC codebase [27].

8 Evaluation

To evaluate Polyphony’s (pol) performance, we ask:

- Does Polyphony converge to meet SLOs?
- How long does convergence take?
- Can Polyphony adapt to changing conditions?
- How does Polyphony’s performance depend on the speed and accuracy of its performance model?
- Which of Polyphony’s components influences its performance the most?

We use CloudLab [16] to evaluate Polyphony on real machines, where performance depends on real transport protocols, NIC hardware, and system-level scheduling noise. This setting exposes the controller to a range of variability present in real deployments, capturing effects that models and simulations omit.

While CloudLab provides a realistic testbed, it is also resource-constrained, especially in the number of bare-metal switches. To test larger topologies with more diverse workloads, we perform a scalability analysis with ns-3 [38] as the ground truth.

8.1 Setup

Classes and knobs. Across experiments, we split traffic into three classes, corresponding to Differentiated Services

Variant	Description	Model acc.	Δt
pol/pmn	Polyphony/Parsimon [55]	Low	~95s
pol/m3	Polyphony/m3 [32]	High	~7s
SelfTune	SelfTune baseline [29, 48]	—	<1ms

Table 1: Controller variants used in the evaluation. Δt is the amount of time it takes to compute the next configuration after receiving feedback from CloudLab.

Code Point (DSCP) values 10, 18, and 26. Each class is assigned its own configuration of three control knobs: switch weight, DCTCP marking threshold K, and initial congestion window (CWND). These knobs span different layers of the stack and together define a 9-dimensional configuration space.

Variants and baseline. Table 1 shows two variants of Polyphony using different performance models and the baseline. pol/pmn uses Parsimon [55] as the performance model. This shows how the controller behaves when the performance model is far slower and less accurate. In Figure 2a, we saw that Parsimon had 62% average error on CloudLab, and in our controller experiments, we observe that each controller iteration takes a minute and a half. Compared to pol/pmn, pol/m3 uses a much more accurate machine learned model m3, which saw 17.6% average error in Figure 2a and reduces the controller iteration time to seven seconds. Lastly, we configure SelfTune as a baseline.

Configuring SelfTune. SelfTune has hyperparameters δ and η , which are the perturbation radius and learning rate, respectively. The authors recommend setting $\delta = O(1)$ and $\eta = O(\delta^2)$, so we choose $\delta = 0.1$ and $\eta = 0.01$. To specify a reward function, we reuse our objective from Section 4.1, but we rescale it to be in $(-1, 1)$ using the hyperbolic tangent function. If g is the value of the objective we’re trying to minimize, we set the reward to $\tanh(-\beta g)$. The parameter β is tuned to typical values of the objective to prevent the reward from saturating too early or responding too weakly. In our experiments, we set $\beta = 0.3$. To make SelfTune tune simplex parameters, we replicate Polyphony’s strategy: we tune the parameters in unconstrained space and then apply softmax to map them back to the simplex. Finally, we have found SelfTune to be sensitive to noisy measurements. To improve its stability, we wait longer for the objective to settle before computing the reward – two minutes vs. the default one minute.

Metrics. We evaluate each controller variant using four key metrics. First, we track the optimization objective over time, which is a unified metric for SLO compliance and fairness. Second, we report *convergence time*, defined as the first point after which the system meets all SLOs for $k = 3$ consecutive iterations. Third, we measure *hindsight regret*, which integrates the difference between the actual objective and the best achievable in hindsight, penalizing exploration that does not improve the objective. Lastly, we compute *min-max fairness* at the end of each experiment, defined as the ratio of the smallest to largest normalized slowdown across classes.

CloudLab setup. We run our experiments on five x1170 machines connected via 10G links to a Dell S4048-ON switch. One machine acts as the manager, while the remaining four serve as traffic endpoints: one receiver and three senders. Each machine runs emu (Section 7), our workload generator, which issues TCP flows between senders and receiver according to specified traffic patterns. This environment includes real NICs, full transport stacks, and operating system-level variability, allowing us to evaluate the controller under realistic sources of noise and nondeterminism.

8.2 Convergence study

We begin by evaluating 1) whether Polyphony can discover configurations that satisfy SLOs and 2) how long convergence takes. This test evaluates convergence against a fixed, known workload with three traffic classes and varying levels of SLO stringency. In Section 8.3, we evaluate Polyphony’s ability to adapt to dynamically changing workloads from Workfeed.

We use publicly released workloads from Meta’s data center network [43]. The flow size distributions are sampled from applications like databases, web servers, and Hadoop. Inter-arrival times follow a log-normal distribution with a $\sigma = 2$ for bursty traffic [55]. Flows are evenly distributed across DSCP classes 10, 18, and 26, each contributing 20% for a network load 60%. Experiments run for 60 minutes under three levels of SLO tightness, specified as thresholds for 99th percentile slowdown—one per DSCP: *Low*: (12, 16, 22), *Medium*: (11, 15, 20), *High*: (10, 13, 17).

Table 2 summarizes convergence time and objectives across low, medium, and high SLO tightness levels. Figure 5 shows the 99th percentile flow completion time slowdowns for each traffic class over time, under high SLO tightness. pol/m3 (Figure 5a) converges within 9.5 minutes, satisfying all per-class SLOs. Its behavior remains stable and consistent across traffic classes. In contrast, pol/pmn (Figure 5b) converges slower due to a slower and less accurate model. While it stabilizes for DSCP 18 and 26, it fails to meet the SLO for DSCP 10. In Figure 5c, SelfTune struggles to meet the SLOs for DSCP 10 and 18, oscillating without converging to better settings over the course of the run. This illustrates the challenge of optimizing under noise and high dimensionality without a performance model to guide search.

Figure 6 presents the global objective evolution under high SLO tightness for three controller variants. pol/m3 improves quickly and stabilizes in 10 minutes. pol/pmn achieves gradual objective reduction, but its trajectory flattens early and does not converge within the 60-minute window due to the slower and less accurate model. SelfTune improves the objective slightly but struggles to extract a meaningful gradient signal from noisy measurements.

Figure 7 shows the evolution of parameters under pol/m3: switch queue weights, initial CWNDs, and DCTCP marking thresholds. All parameters gradually adjust and remain stable over time. We observe that Polyphony jointly optimizes host

Variant	Converge Time	Regret	Fairness	Final Obj.
Low SLO tightness				
pol/m3	3.7 mins	21.0	0.83	1.49
pol/pmn	38.2 mins	67.7	0.66	2.01
SelfTune	Not converged	89.7	0.57	3.30
Medium SLO tightness				
pol/m3	6.0 mins	20.7	0.86	1.43
pol/pmn	Not converged	83.2	0.72	2.09
SelfTune	Not converged	127.7	0.46	3.15
High SLO tightness				
pol/m3	9.5 mins	24.7	0.89	1.47
pol/pmn	Not converged	79.9	0.70	2.44
SelfTune	Not converged	154.4	0.47	3.99

Table 2: Convergence metrics across SLO tightness levels in CloudLab.

Phase	DSCP 10	DSCP 18	DSCP 26
I: 0–30 min	500	2000	3500
II: 30–60 min	3500	2000	500
III: 60–90 min	500	2000	3500
IV: 90–120 min	3500	2000	500

Table 3: Traffic load profiles used in the adaptation study. Offered load (in Mbps) varies every 30 minutes to challenge controller responsiveness.

and network settings to balance fairness and performance. Note that the optimal initial window size depends on the scheduling weight, so that the medium traffic class gets the largest window size. A large initial window with high weight will negatively impact the SLOs for other traffic classes.

8.3 Adaptation study

This section evaluates whether Polyphony can adapt its control decisions in the presence of workload changes measured live from Workfeed.

In this experiment, we use the same bursty traffic workloads as described in Section 8.2, and we impose a uniform SLO of 13.5 on all classes. To test adaptability, we introduce three regime shifts during a two hour experiment, and we use Workfeed to measure the workload and send it to the controller. Table 3 summarizes the offered load profile across four 30-minute phases. Each phase requires the controller to reconfigure both host and switch-level parameters to maintain per-class SLO compliance.

Figure 8 shows the per-class 99th percentile flow slowdowns throughout the adaptation experiment. pol/m3 (Figure 8a) adapts quickly to workload shifts, reestablishing SLOs and equalizing classes within minutes. pol/pmn (Figure 8b) adapts slower, taking nearly a half hour in some cases to reach the SLO threshold. It does not converge fast enough to equalize classes within any half hour window. SelfTune (Figure 8c) does little to improve performance and does not respond to workload changes.

Figure 9 shows the global objective over time. pol/m3 quickly reduces the objective quickly after each workload shift. While pol/pmn steers away from large SLO violations,

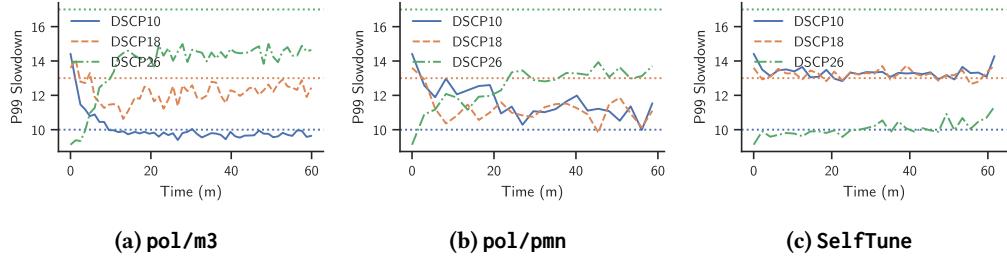


Figure 5: Per-class 99th percentile flow completion time slowdowns under high SLO constraints. *pol/m3* meets all SLOs within a few iterations. *pol/pmn* slowly trends toward SLO conformance but does not meet the SLO for DSCP 10. Finally, *SelfTune* misses SLOs for two of the three classes, showing little improvement when relying on noisy gradient estimates.

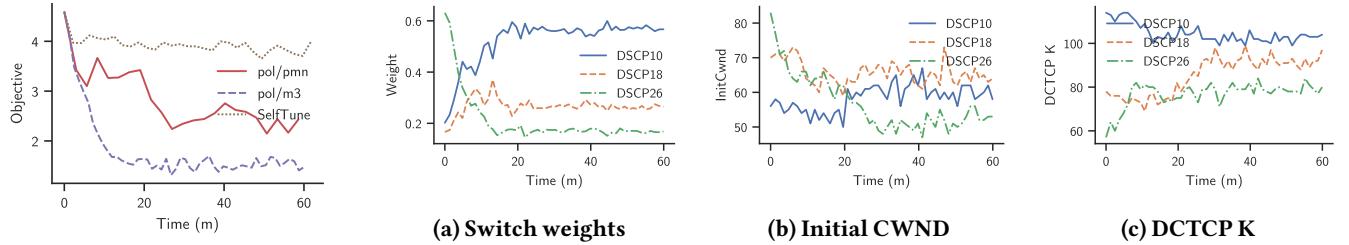


Figure 6: Objective over time under high SLO constraints.

Figure 7: Parameter trajectories for *pol/m3* under high SLO constraints. Polyphony coordinates switch and host tuning to meet SLOs and fairness objectives.

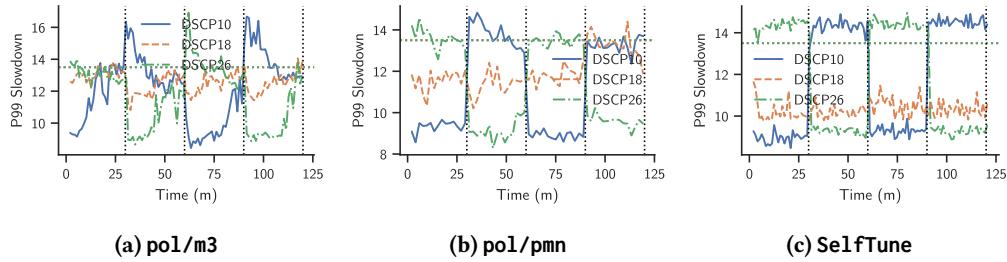


Figure 8: Per-class 99th percentile flow completion time slowdowns under workload shifts. *pol/m3* adapts within minutes to reestablish SLO compliance. *pol/pmn* converges slowly and has a higher proportion of missed SLOs. *SelfTune* does not meet SLOs and shows no clear adaptation.

it scores poorly in the fairness dimension, so the objective is persistently higher. *SelfTune* does not reduce the objective throughout the experiment.

Lastly, Figure 10 shows how *pol/m3*'s parameter evolution over time. Most interpretable are the switch weights (Figure 9a), which shift predictably to favor under-served classes in each regime (e.g., DSCP 10 in Phase II).

8.4 Ablation study

We use the same bursty workload and high tightness SLO thresholds as in Section 8.2. All experiments are based on *pol/m3*, but each omits a specific component:

- *No trust region (TR)* disables the trust region.
- *No correction* disables the online model correction, relying solely on *m3*'s predictions.

- *No model* removes the model (*m3*), forcing the controller to explore directly on the real system.
- *No denoiser* feeds raw (unsmoothed) slowdown measurements to the controller.

Figure 11 presents per-class slowdown trajectories and corresponding weight adjustments. Removing the trust region (Figure 11a) results in larger parameter changes and early SLO violations (especially for DSCP 18 and 26), with weight oscillations between extremes. Without residual correction (Figure 11b) the biased simulator mis-orders nearby points, and fails to improve the objective within a trust region. Without the model (Figure 11c), the controller must explore many more configurations before it finds a good one, missing SLOs for the entire duration. The “no denoiser” variant performs well but exhibits high-frequency jitter in slowdowns and weights.

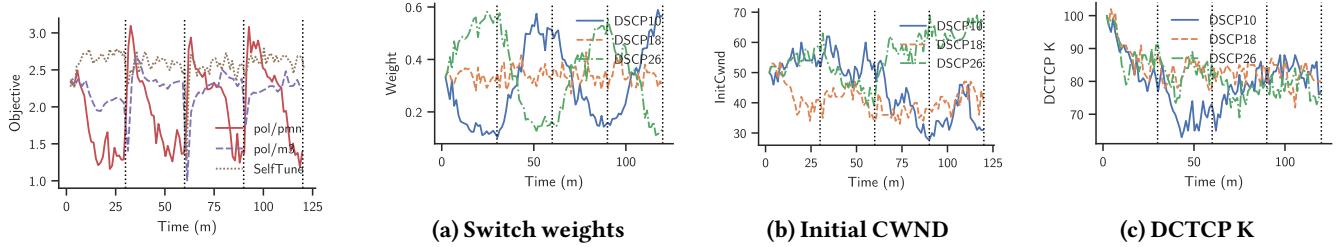


Figure 9: Objective over time under regime shifts.

Figure 10: Parameter evolution under pol/m3. Polyphony gradually updates parameters in response to workload changes.

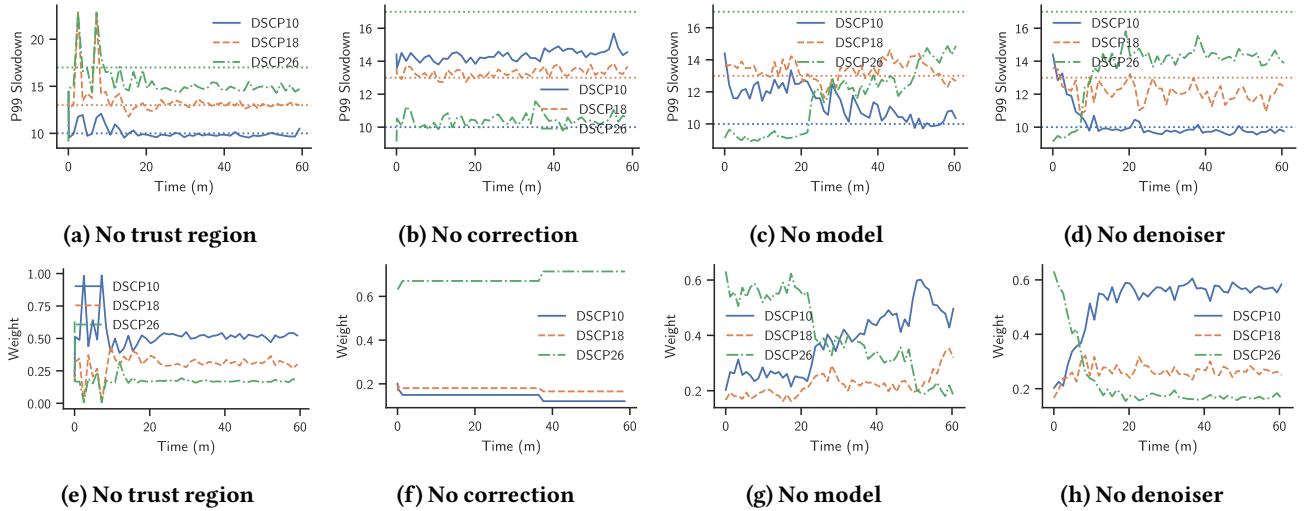


Figure 11: Per-class 99th percentile slowdowns (top) and switch weights (bottom) under ablation. Removing key components—like the trust region, performance model, or correction—degrades SLO compliance and stability. Note: Y-axis scales vary.

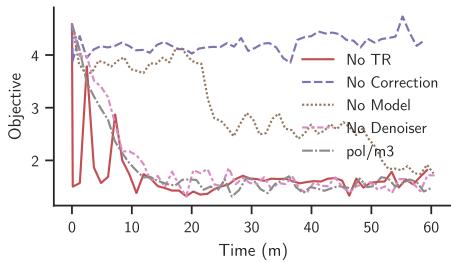


Figure 12: Global objective over time under component ablations. The full Polyphony controller (pol/m3) and no denoiser both converge; no trust region converges but with oscillation. No model converges more slowly, and no correction does not make progress.

Figure 12 summarizes the global objective under each variant. pol/m3 converges smoothly. "No model" and "no correction" both plateau early, with the latter performing the worst overall. "No TR" reduces the objective, but with high initial variance.

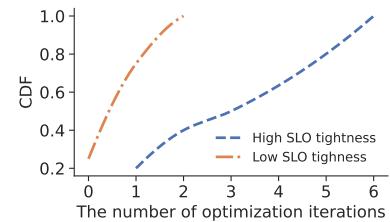


Figure 13: CDF of convergence time across ns-3 scenarios; scenarios that do not converge are omitted.

8.5 Scalability analysis in ns-3

While our CloudLab [16] experiments validate Polyphony in realistic and noisy conditions, they are limited to a small topology. We conduct a scalability analysis using ns-3 [38].

We construct 24 simulation scenarios by varying the following parameters:

- **Topology:** We use a 32-rack, 256-host fat-tree topology sampled from Meta’s data center networks [6].
- **Workload:** We generate flow-level traffic using two flow size distributions derived from publicly available production workloads (WebSearch and Hadoop [43]). Inter-arrival times follow a log-normal distribution with $\sigma = 1$ (low

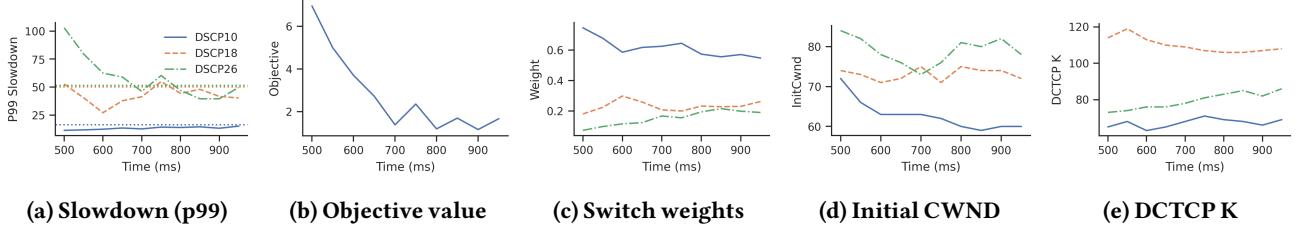


Figure 14: pol/m3’s behavior in ns-3 [38]. (a) Per-class 99th percentile flow completion time slowdowns; dotted lines indicate SLO thresholds. (b) Global objective improves steadily over time. (c–e) Controller updates switch weights, initial congestion windows, and DCTCP thresholds smoothly to meet SLOs.

burstiness) or $\sigma = 2$ (high burstiness) [55]. The network load is fixed at 60%. We use three application-specific traffic matrices, database, web server, and Hadoop clusters [43], to capture realistic rack-to-rack communication patterns. Flow endpoints are selected uniformly at random from hosts within each rack.

- **SLO tightness:** For each scenario, we randomly select an SLO tightness level by scaling a baseline slowdown by 25% (high tightness) or 50% (low). Baseline slowdowns are obtained by running a simulation under a fixed, near-optimal configuration. The 99th percentile slowdown from this simulation is used as the baseline SLO and scaled accordingly.

We evaluate pol/m3 based on two metrics: *SLO compliance rate* and *convergence time* across all scenarios. Each simulation runs for 1 second. The first 500 ms are used to gather an initial set of datapoints. Starting at 500 ms, pol/m3 selects a new configuration every 50 ms.

Figure 14 visualizes a representative scenario: the controller reduces 99th percentile flow slowdowns (a) and global objective (b) while making coordinated updates to switch weights (c), initial CWNDs (d), and DCTCP thresholds (e). Figure 13 shows the CDF of convergence times across the 19 scenarios where pol/m3 successfully converged. pol/m3 achieves 80% (19/24) SLO compliance. Despite variations in workload characteristics and SLO tightness, pol/m3 stabilizes to SLOs in most cases: under low SLO tightness, almost all scenarios converge within two optimization iterations; under high SLO tightness, convergence takes longer but still completes within six iterations in all cases.

9 Related Work

9.1 Tuners and controllers

Blackbox autotuners. A large body of work automates configuration tuning by treating the system as a black box and running controlled trials to search high-dimensional knob spaces. Representative systems include CherryPick [3] for cloud instance choices, Metis [34] and OtterTune [50] for service/DBMS parameters, OPPerTune [48] and SelfTune [29] for post-deployment service and cluster-manager knobs, AutoSched [22] for deep-learning schedulers, and FLASH [42] as

a sample-efficient Bayesian optimization method. These approaches are supervisory and episodic: they propose a configuration, measure the live system, and iterate to convergence—often over minutes to hours—and can induce temporary regressions during exploration. They are effective for finding good static settings but are not designed to regulate a performance metric on a tight feedback loop.

Online controllers. A separate line uses continuous feedback to control a system variable directly. For example, Autothrottle employs a bi-level design with a contextual-bandit making minute-by-minute CPU-throttle targets for microservices to satisfy latency SLOs [51], and ACC applies deep RL to adjust ECN thresholds at switches to keep queues short while sustaining throughput [52]. These are online regulators of specific mechanisms (CPU throttling; ECN marking) rather than broad configuration search. Polyphony sits in this controller category: it targets network QoS metrics directly (e.g., class-specific p99) and selects actions on a minute timescale, but, unlike prior online controllers, does so model-guided, using fast approximate performance models to predict counterfactual outcomes and reduce the need for risky live experimentation.

9.2 Simulation-based optimization and control

Simulation offers a scalable and low-risk means to explore system behavior, especially in complex networking environments. Modern network simulators use parallelization [7, 21, 55] and ML techniques [18, 32, 53, 54] to model packet-level behavior with improved speed and fidelity. Despite these advances, a gap remains between simulated and real-world performance due to unmodeled hardware interactions and environmental variability [19]. To bridge this gap, ByteDance’s Crescent [23] emulates production switch software in isolated environments, enabling realistic and low-risk evaluation of network configurations. Using simulations and emulations as predictive models, Model Predictive Control (MPC) offers formal methods for proactive configuration planning. Zipper [9], for instance, applies MPC to dynamically allocate bandwidth in 5G RAN slicing, using model-based forecasts to maintain SLO compliance. However, conventional MPC approaches require sufficiently accurate predictions; robust variants exist,

but building fast, globally accurate models in data center networks is an open problem. Polyphony addresses this problem with an online corrective overlay to compensate for modeling errors, and it further applies trust-region-based optimization strategies to focus exploration in regions where predictions are most reliable.

10 Conclusion

This paper described the design, implementation, and evaluation of Polyphony, a system for controlling tail latency at minute-scale to achieve performance objectives for different traffic classes—even under dynamic workloads. Polyphony leverages fast but inaccurate models to estimate the consequence of applying a proposed configuration change. To correct for model error, Polyphony builds and applies a corrective overlay anchored on the best known alternative; Bayesian optimization is then used within a region around that alternative. Provided that the prediction model is reasonably accurate, e.g., using machine learning, Polyphony can quickly converge to a more optimal operating point and adapt to changing conditions. Under tight SLOs, noisy measurements, and large workload shifts, Polyphony stabilizes to meet SLOs within fifteen minutes.

References

- [1] J. Aitchison. The statistical analysis of compositional data. *Journal of the Royal Statistical Society: Series B (Methodological)*, 44(2):139–160, 1982.
- [2] A. G. Alcoz, A. Dietmüller, and L. Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, 2020.
- [3] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the USENIX NSDI*, pages 469–482, 2017.
- [4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
- [5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, 2012.
- [6] A. Andreyev. Introducing data center fabric: The next-generation facebook data center network, 2014. Facebook Engineering Blog.
- [7] S. Bai, H. Zheng, C. Tian, X. Wang, C. Liu, X. Jin, F. Xiao, Q. Xiang, W. Dou, and G. Chen. Unison: a parallel-efficient and user-transparent network simulation kernel. In *Proceedings of the European Conference on Computer Systems*, pages 115–131, 2024.
- [8] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, Apr. 2023. USENIX Association.
- [9] A. Balasingam, M. Kotaru, and P. Bahl. {Application-Level} service assurance with 5g {RAN} slicing. In *Proceedings of the USENIX NSDI 24*, pages 841–857, 2024.
- [10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An Architecture for Differentiated Services, 1994.

- [11] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [12] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated Services in the Internet Architecture: an Overview, 1994.
- [13] E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010.
- [14] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures & Protocols*, pages 14–26, 1992.
- [15] R. Cocchi, D. Estrin, S. Shenker, and L. Zhang. A Study of Priority Pricing in Multiple Service Class Networks. In *Proceedings of the ACM SIGCOMM Conference on Communications Architecture & Protocols*, SIGCOMM ’91, page 123–130, New York, NY, USA, 1991. Association for Computing Machinery.
- [16] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of {CloudLab}. In *Proceedings of the USENIX ATC*, pages 1–14, 2019.
- [17] D. Eriksson, M. Pearce, J. Gardner, R. D. Turner, and M. Poloczek. Scalable global optimization via local bayesian optimization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [18] M. Ferriol-Galmés, J. Paillisse, J. Suárez-Varela, K. Rusek, S. Xiao, X. Shi, X. Cheng, P. Barlet-Ros, and A. Cabellos-Aparicio. Routenet-fermi: Network modeling with graph neural networks. *IEEE/ACM transactions on networking*, 31(6):3080–3095, 2023.
- [19] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the International Conference on Software Engineering*, pages 299–310, 2014.
- [20] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohita, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.
- [21] K. Gao, L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the SIGCOMM*, pages 167–181, 2023.
- [22] W. Gao, X. Zhang, S. Huang, S. Guo, P. Sun, Y. Wen, and T. Zhang. Autosched: An adaptive self-configured framework for scheduling deep learning training workloads. In *Proceedings of the ACM International Conference on Supercomputing*, pages 473–484, 2024.
- [23] Z. Gao, A. Abhashkumar, Z. Sun, W. Jiang, and Y. Wang. Crescent: emulating heterogeneous production network at scale. In *Proceedings of the USENIX NSDI*, pages 1045–1062, 2024.
- [24] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, 2022.
- [25] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don’t matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.
- [26] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.
- [27] High-Precision-Congestion-Control . <https://github.com/alibaba-edu/High-Precision-Congestion-Control>, 2019.
- [28] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.
- [29] A. Karthikeyan, N. Natarajan, G. Somashekhar, L. Zhao, R. Bhagwan, R. Fonseca, T. Racheva, and Y. Bansal. {SelfTune}: Tuning cluster managers. In *Proceedings of the USENIX NSDI*, pages 1097–1114, 2023.
- [30] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 514–528, 2020.
- [31] R. Lafage. egobox, a rust toolbox for efficient global optimization. *Journal of Open Source Software*, 7(78):4737, 2022.
- [32] C. Li, A. Nasr-Esfahany, K. Zhao, K. Noorbakhsh, P. Goyal, M. Alizadeh, and T. E. Anderson. m3: Accurate flow-level performance estimation using machine learning. In *Proceedings of the ACM SIGCOMM*, pages 813–827, 2024.

- [33] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, page 44–58, 2019.
- [34] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly tuning tail latencies of cloud systems. In *Proceedings of the USENIX ATC*, pages 981–992, 2018.
- [35] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.
- [36] J. C. Mogul and J. Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 136–141, 2019.
- [37] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.
- [38] ns-3 Network Simulator. <https://www.nsnam.org>, 2020.
- [39] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery.
- [40] L. Poutievski, O. Mashayekhi, J. Ong, A. Singh, M. Tariq, R. Wang, J. Zhang, V. Beauregard, P. Conner, S. Gribble, R. Kapoor, S. Kratzer, N. Li, H. Liu, K. Nagaraj, J. Ornstein, S. Sawhney, R. Urata, L. Vicisano, K. Yasumura, S. Zhang, J. Zhou, and A. Vahdat. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.
- [41] Prometheus Authors. Prometheus: Monitoring system and time series database. <https://prometheus.io>, 2024. Accessed: 2025-04-24.
- [42] H. Qiu, W. Mao, A. Patke, S. Cui, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. Flash: Fast model adaptation in ml-centric cloud platforms. *Proceedings of Machine Learning and Systems*, 6:524–544, 2024.
- [43] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.
- [44] K. Seemakhupt, B. E. Stephens, S. Khan, S. Liu, H. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 498–514, New York, NY, USA, 2023. Association for Computing Machinery.
- [45] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman. Programmable Calendar Queues for High-Speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, 2020.
- [46] S. Shenker, C. Partridge, and R. Guerin. RFC 2212: Specification of Guaranteed Quality of Service, 1997.
- [47] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, page 183–197, 2015.
- [48] G. Somashekhar, K. Tandon, A. Kini, C.-C. Chang, P. Husak, R. Bhagwan, M. Das, A. Gandhi, and N. Nataraajan. {OPPerTune}:{Post-Deployment} configuration tuning of services made easy. In *Proceedings of the USENIX NSDI*, pages 1101–1120, 2024.
- [49] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. *SIGCOMM Comput. Commun. Rev.*, 28(4):118–130, Oct. 1998.
- [50] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the ACM international conference on management of data*, pages 1009–1024, 2017.
- [51] Z. Wang, P. Li, C.-J. M. Liang, F. Wu, and F. Y. Yan. Autothrottle: A practical {Bi-Level} approach to resource management for {SLO-Targeted} microservices. In *Proceedings of the USENIX NSDI*, pages 149–165, 2024.
- [52] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng. Acc: Automatic ecn tuning for high-speed datacenter networks. In *Proceedings of the ACM SIGCOMM*, pages 384–397, 2021.
- [53] Q. Yang, X. Peng, L. Chen, L. Liu, J. Zhang, H. Xu, B. Li, and G. Zhang. DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-Level Visibility. In *Proceedings of the ACM SIGCOMM*, pages 441–457, 2022.
- [54] Q. Zhang, K. K. Ng, C. Kazer, S. Yan, J. Sedoc, and V. Liu. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pages 287–304, 2021.
- [55] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson. Scalable tail latency estimation for data center networks. In *Proceedings of the USENIX NSDI*, pages 685–702, 2023.
- [56] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and

M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM*

2015 Conference, page 523–536, 2015.