

Worst-Case Discovery and Runtime Protection for RL-Based Network Controllers

Hongyu Hè*
Princeton

Minhao Jin
Princeton

Maria Apostolaki
Princeton

Abstract

RL-based controllers achieve strong average-case performance in networking tasks such as congestion control and adaptive bitrate streaming. Yet their performance can degrade severely under network conditions where strong performance is still achievable. Identifying such conditions and quantifying the resulting performance gap is intractable by enumeration, while the sequential and closed-loop nature of RL controllers makes formal verification methods impractical.

We present REGUARD, a framework that discovers worst-case scenarios for a given RL controller and protects it against them at inference time without retraining. Discovery is formulated as a bilevel regret-maximization problem, which yields a certified lower bound on the worst-case performance gap. The discovered trajectories are then analyzed as counterfactuals and compiled into lightweight logic rules that intervene only when a risky state is detected, leaving the controller’s behavior unchanged otherwise.

We evaluate REGUARD across three RL-based network controllers: Pensieve, Sage, and Park. REGUARD discovers scenarios in which the controller’s performance is 43–64% worse than what is achievable. REGUARD not only discovers gaps 57% to 6× larger than those found by the strongest baselines, but also shrinks them by 79–85% via lightweight rule-based protection while preserving nominal performance. REGUARD’s protection extends beyond the scenarios it discovers, improving performance across a wider range of network conditions.

1 Introduction

For more than a decade, RL-based controllers have been proposed for networked systems, *e.g.*, adaptive bitrate streaming, congestion control, and resource management [3, 26, 38, 39]. These controllers have been shown to dramatically outperform the best handcrafted heuristics, making a compelling case for wide adoption. Yet before an operator can justify deploying

one in production, she needs answers to questions that benchmarks alone cannot provide. Are there scenarios, *i.e.*, sequences of network conditions, under which the RL controller performs far worse than what is achievable? If so, how much performance does it forfeit? Can such performance gaps be mitigated without sacrificing the controller’s performance gains on other, perhaps more common, scenarios?

Answering these questions is fundamentally hard. First, an RL controller is a black-box neural policy that is hard to understand and modify. Second, the space of all possible network scenarios is exponentially large. Finally, the controller’s decisions and the network’s dynamics are coupled in a closed loop, where each affects the evolution of the other over time. This complexity makes verification and formal analysis methods [12, 31] impractical because they require a formal model. Approximating RL controllers with interpretable or differential counterparts [18, 29] might offer some insights for manual inspection, but the loss of fidelity during the approximation prevents them from reliably identifying worst-case scenarios. Finally, approaches that combine training with scenario generation or selection, such as Genet [38] and Mowgli [4], aim to improve average performance and are thus orthogonal to our goal. In fact, we find that adapting them to target worst-case scenarios does not reliably protect the target controller against them and often degrades performance on others.

This paper presents REGUARD, a practical framework that addresses all three questions without requiring a formal model of the controller or the network dynamics. Rather than accumulating scenarios and hoping that training will fix everything, REGUARD strategically focuses on discovering and correcting scenarios of maximum controller regret, meaning scenarios where the controller performs poorly but strong performance is still achievable. Beyond the vastness of the scenario space, determining the best achievable performance for each scenario is itself a hard optimization problem. To address this, REGUARD casts scenario discovery as bilevel regret maximization over feasible network conditions. The outer optimization searches for the scenario that maximizes the performance gap of the targeted controller, while the inner optimization identifies the

*Correspondence to hhy@g.princeton.edu.

best available strategy for that scenario. For the inner loop, REGUARD selects among the actions of a portfolio of existing heuristics, *i.e.*, established algorithms that each perform well in different operating regimes. Beyond making the inner optimization tractable by reducing it to selection over a finite set of known policies, the portfolio serves two additional purposes. First, it makes the lower bound on worst-case regret certified by the bilevel optimization meaningfully tight, since the bound’s quality depends on how closely the portfolio approximates the true optimum, and decades of networking research have produced heuristics that are individually strong and collectively cover diverse operating regimes. Second, it provides a practical counterfactual, revealing what a better strategy would have done under the same conditions and in which direction the controller’s decisions should be corrected. REGUARD then analyzes the discovered counterfactual trajectories, *i.e.*, scenarios and controller actions, to derive recurring patterns that precede the controller forfeiting performance. It compiles these patterns into logic rules that are evaluated only against the controller’s current observable state and intervene only when a risky pattern is detected. This design preserves the controller’s nominal performance, which retraining-based approaches consistently sacrifice, while providing targeted protection in failure-prone regimes.

We demonstrate REGUARD’s benefits and portability across three state-of-the-art RL controllers for distinct networking tasks: Pensieve for adaptive bitrate streaming, Sage for congestion control, and Park for load balancing. REGUARD discovers scenarios in which these controllers achieve 43–64% worse performance than a near-optimal reference, exposing gaps orders of magnitude larger than those found by alternative discovery methods such as Genet [38], Indago [8], and Gilad et al. [13]. With REGUARD’s run-time protection, these gaps shrink by up to 79–85%.

Critically, REGUARD’s protection extends well beyond the specific scenarios it discovers, confirming that it targets genuine controller vulnerabilities rather than overfitting to very particular conditions. Indeed, REGUARD-protected controllers also outperform their unprotected counterparts in scenarios identified by alternative approaches (*e.g.*, Genet, Indago, and Gilad et al.), even though those scenarios were never used as counterfactuals. Furthermore, re-running REGUARD’s discovery against the REGUARD-protected controller yields substantially smaller worst-case regret, demonstrating that REGUARD shrinks the controller’s overall vulnerability surface. Finally, REGUARD fits comfortably within each controller’s inference-time budget, confirming that its protection can run alongside the controller without delaying its decisions. Thanks to the efficient merging of patterns and the use of predicates over raw observable state variables, the rules are interpretable enough for manual inspection, allowing us to validate that they capture meaningful controller weaknesses. For instance, REGUARD’s rules expose that Pensieve consistently selects bitrates far above what the available bandwidth can sustain, confirming

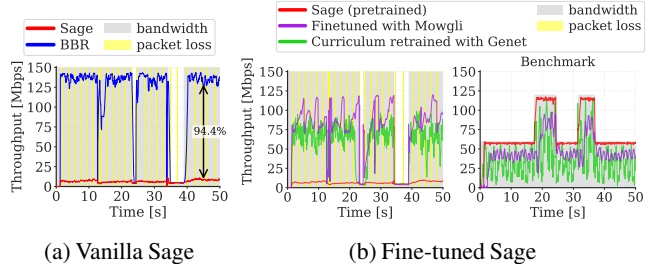


Figure 1: (a) shows a concrete scenario where Sage [40] has 94% less throughput than what BBR can achieve under the same condition, despite its superiority in other scenarios. (b) shows that fine-tuning Sage for protection does not lift its throughput to the achievable level in that scenario and also degrades Sage’s performance on a benchmark where it performed well before fine-tuning.

a systematic tendency to overshoot under scarce conditions.

2 Motivation

This section uses RL-based congestion control as an example use case to explain why strong benchmark performance is insufficient for real-world RL deployment in networked systems and to derive the requirements for REGUARD. We detail the use cases considered in Table 1 and Appendix B.

2.1 Use Case: RL-Based Congestion Control

Consider an operator managing a private datacenter network. To improve performance for latency-sensitive and high-throughput workloads, she considers replacing the currently deployed BBR with Sage [40], a state-of-the-art RL-based congestion controller that has been shown to achieve both higher throughput and lower latency. She verifies these claims on representative internal traces, and the results are encouraging. But encouraging is not enough to justify a production deployment. The operator needs to know: **How much can Sage deviate from the optimal achievable performance?**

Her first instinct is formal verification. Tools like MetaOpt [31] and whiRL [12] can provide reliable worst-case guarantees, but they would require her to model, in logic, every component of the system: the neural policy that maps observations to sending-rate decisions, how those decisions determine packets in flight and thus queue occupancy, how queue occupancy in turn drives loss and inflates RTT, how the achieved throughput feeds back into the controller’s next observation, and how all of these quantities are simultaneously shaped by the network conditions themselves, such as available bandwidth, propagation delay, and background traffic, independently of anything the controller does. For a system as complex as Sage, accurate modeling is prohibitively difficult, and any approximation risks producing scenarios that mask the true worst case.

Her next thought is adversarial ML, which should, in principle, be well suited to stress-testing neural policies. But adversarial methods such as PGD (projected gradient descent [9, 14, 23, 25]) operate directly on the controller’s input features (shown in Table 2) without respecting the causal dependencies between them. As a result, PGD could, for instance, perturb Sage’s delivery-rate and loss features in a way that implies near-capacity throughput and severe persistent loss, a combination that is inconsistent with feasible queuing and ACK dynamics under the same bottleneck link. Hence, the resulting scenario may not just be unlikely, but impossible in practice. Worse, the operator has no systematic way to check which scenarios are possible. This issue is especially acute for RL-based network controllers. In standard feed-forward models, one can often encode feature dependencies with moderate effort [7, 20, 34]. In contrast, RL controllers operate sequentially: current inputs depend on past actions, and future states depend on both controller decisions and network dynamics. Capturing these dependencies would require modeling the entire interactive system, bringing our operator back to the same complexity barrier faced by formal analysis.

The remaining option is random testing. But the search space is exponentially large in the number of control steps, making exhaustive exploration impossible. Running random tests long enough will eventually surface scenarios where Sage underperforms BBR, but there is no guarantee that the discovered scenarios are anywhere close to the true worst case. She is effectively left with anecdotes rather than evidence and with no basis for quantifying the deployment risk she is actually taking on.

Even if the operator could somehow discover scenarios where Sage underperforms BBR, such as the one shown in Fig. 1a, she still needs a way to mitigate them before enjoying the benefits of deploying Sage. The canonical approach would be fine-tuning and/or curriculum-based retraining [21, 22, 24, 36, 37], which have also gained popularity in the networking community. Unfortunately, these approaches induce global policy updates. They alter behavior globally rather than only in specific worst-case scenarios, moving the policy away from behaviors that worked well under normal conditions. As an illustration, Fig. 1b shows the performance of Sage after it has been fine-tuned on more scenarios, including the scenario of Fig. 1a. Although its throughput improves, it still does not reach the achievable level. Worse, its performance degrades on another scenario where it previously performed well. Our operator now needs to decide: **Should I optimize Sage for good average performance or for robustness in the worst case?**

2.2 Requirements & Design Principles

The example above highlights the need for a system that does not replace or compete with RL-based network controllers, but instead complements them to improve their trustworthiness, robustness, and ultimately their deployability.

First, such a system must be able to discover network

scenarios, namely sequences of network conditions, under which the target controller performs substantially worse than what is achievable under the same conditions. This captures the true worst case for the controller: not merely a challenging network, but a setting where the controller fails to adapt despite strong performance still being possible. We cannot control how harsh the environment is, but we can identify when the controller responds poorly to it. For instance, the scenario in Fig. 1a where Sage achieves 94% lower throughput than BBR is precisely the type of embarrassing failure an operator would want to know about and have fixed before deployment.

Second, a system that works alongside an RL controller must preserve the controller’s strong nominal performance. Any mitigation that sacrifices the controller’s nominal performance defeats the purpose of deploying it. As Fig. 1b illustrates, fine-tuning Sage on challenging scenarios degrades its throughput on normal conditions, trading one problem for another.

Finally, the system must remain practical to deploy. In particular, it should not require formally modeling the controller, the environment, or their interaction dynamics in advance. Such modeling is prohibitively complex for modern RL-based networked systems and would undermine usability in practice.

3 Overview

Following these design principles, REGUARD takes the form shown in Fig. 2. Next, we highlight the main insights that drove its design.

By framing discovery as regret maximization over a bilevel program, we can provide a tightness guarantee on how close the discovered failure is to the true worst case. To find scenarios that leave the most achievable performance on the table, we need to guide the search towards worst-case regret, namely, the performance gap between what the controller does and what is achievable. REGUARD frames discovery as an optimization over all feasible network conditions for the scenario that maximizes regret. Because this is optimization rather than sampling, the returned scenario comes with a formal bound (Theorem 1): the gap between the regret it achieves and the true worst-case regret is bounded by how good the reference policy we have available is. Hence, for the bound to be meaningful, we need a good approximation of the optimal policy. Computing the exact optimal policy is, however, intractable for a stochastic dynamical system over long horizons. Fortunately, decades of networking research have produced a rich set of heuristics that each perform well in different operating regimes. Their per-scenario best (*i.e.*, the upper envelope) serves as a strong proxy for the scenario-specific optimum. To unlock this power, we use a bilevel structure where the inner problem selects the best reference for each candidate environment that the outer problem finds.

The same upper envelope that certifies the failure also provides the counterfactual signal needed for protection.

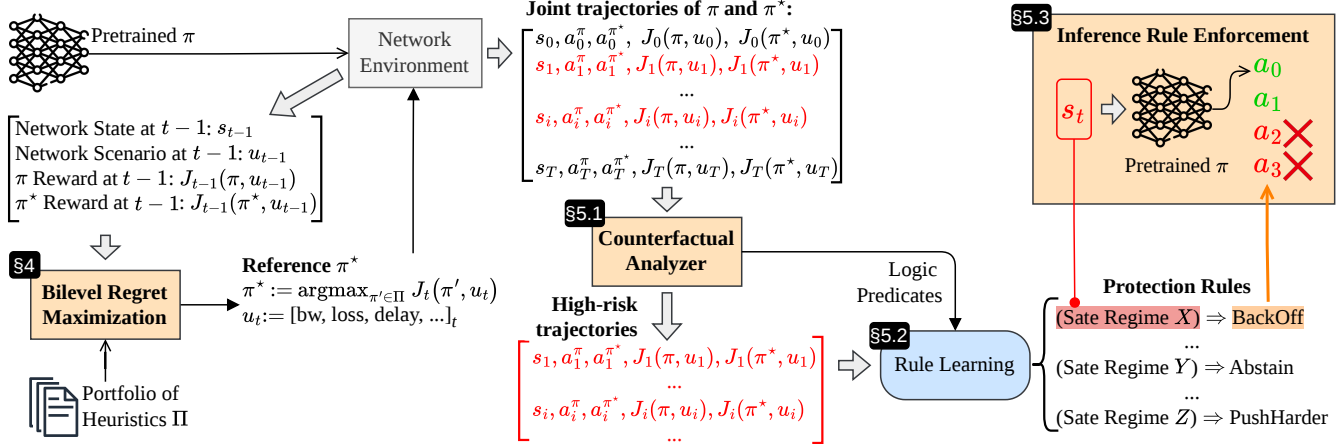


Figure 2: REGUARD discovers scenarios that maximize the targeted RL controller’s regret by directly interacting with the network environment in which the controller was trained and by using a portfolio of heuristics to approximate the scenario-specific optimum. It then analyzes the resulting counterfactuals to extract recurring risky patterns and corrective directions. Finally, REGUARD compiles those counterfactuals into logic rules that override the controller’s decision at inference time only when intervention is strictly necessary.

The portfolio of heuristics serves a second, entirely distinct role beyond certification. When REGUARD discovers a high-regret scenario, it knows not only that the controller failed, but also which heuristic succeeded and what that heuristic did differently at each decision point. This is a counterfactual: under the same network conditions, the reference chose to back off while the controller pushed harder, or vice versa. By analyzing the controller and reference trajectories, REGUARD can identify specific risky states and the corrective directions needed to avoid the performance degradation. Because these high-regret scenarios are used to improve the controller, REGUARD does not rely on a single worst-case scenario, but instead seeks multiple scenarios that expose different blind spots.

The highest-regret scenarios cluster around a few recurring and high-risk decision-making flaws, making surgical intervention both possible and effective. The highest-regret scenarios are not random; they are clear manifestations of systematic flaws in the controller’s policy. Fixing those flaws could, in principle, protect the controller while minimally affecting its performance. Still, because the search objective rewards the same controller weakness every time it surfaces, many of these scenarios expose the same underlying mistake. For instance, REGUARD scenarios often reveal that Park systematically routes small jobs to slow servers even when fast servers sit idle. We observe that the discovered scenarios naturally cluster around a small number of recurring patterns. By mining these patterns, REGUARD extracts a compact set of logic rules over interpretable predicates that cover a wide range of failure conditions while remaining fully auditable by the operator.

4 Bilevel Regret Maximization over Stochastic Dynamical Systems

4.1 Problem Formulation

REGUARD searches for a feasible network scenario that maximizes the regret of a fixed pretrained controller relative to the best reference policy under that same scenario. The input is a pretrained policy π , a reference policy class Π , and a feasible family of network scenarios \mathcal{E} .

Scenario, reward, and regret. A scenario is a finite resource sequence $e = (u_0, \dots, u_{T-1}) \in \mathcal{E}$, where each $u_t \in \mathcal{U}$ specifies the resource available at time t . Thus, e is a time-varying network condition rather than a static input. For any policy π' , let $J(\pi'; e) := \mathbb{E}[\sum_{t=0}^{T-1} \gamma^t r_t | \pi', e]$ denote its trajectory reward in scenario e . The reference policy is scenario-specific: $\pi_e^* \in \operatorname{argmax}_{\pi' \in \Pi} J(\pi'; e)$. The regret of the pretrained controller in scenario e is then $R(e) := J(\pi_e^*; e) - J(\pi; e)$.

This regret isolates genuine controller failures. If a scenario is intrinsically bad for every policy, then both terms are low and the regret stays small. A large regret instead means that the controller leaves substantial reward on the table in a feasible scenario where much better performance is still achievable.

Feasible search space. We restrict the outer search to

$$\mathcal{E} := \{e : e \models c_j, j = 1, \dots, m\}, \quad (1)$$

where the constraints c_j encode the family of network conditions the operator cares about. These constraints may bound resource magnitude, rates of change, queueing behavior, or other deployment-specific requirements.

Bilevel regret maximization. REGUARD targets the follow-

ing bilevel problem:

$$e^* \in \operatorname{argmax}_{e \in \mathcal{E}} [J(\pi_e^*; e) - J(\pi; e)], \quad (2)$$

or equivalently,

$$\begin{aligned} & \operatorname{maximize}_{e \in \mathcal{E}} && J(\pi_e^*; e) - J(\pi; e) \\ & \text{subject to} && \pi_e^* \in \operatorname{argmax}_{\pi' \in \Pi} J(\pi'; e). \end{aligned} \quad (3)$$

The outer level searches over feasible scenarios, while the inner level picks the strongest reference policy for each candidate scenario. A high-scoring scenario is therefore a concrete failure case in which the controller underperforms by a large and avoidable amount.

This formulation is inherently trajectory-level. The outer variable e is a time series of resources, and both $J(\pi; e)$ and $J(\pi_e^*; e)$ depend on the full closed-loop evolution over the horizon. Unlike prior static-input analyses such as MetaOpt [31], REGUARD searches over time-varying network scenarios for dynamical controllers.

Outer RL solver and practical reference. We use RL only as a solver for the outer search over scenarios. This outer solver is distinct from the control policies π and π_e^* , which act *within* a fixed scenario. RL is a natural fit because the scenario itself is sequential, and because the search does not require an explicit formal model of the pretrained controller or of the network dynamics.

In practice, the exact inner solution may be unattainable, especially in continuous-action settings. We therefore use an approximate reference policy $\hat{\pi}_e^* \in \Pi$ and solve

$$\hat{e} \in \operatorname{argmax}_{e \in \mathcal{E}} [J(\hat{\pi}_e^*; e) - J(\pi; e)]. \quad (4)$$

The theoretical target is Eqn. (3), while the implemented procedure approximately solves Eqn. (4).

4.2 Guarantees

The formulation provides a simple guarantee with a clear interpretation. The scenario returned by REGUARD is a certificate that the pretrained controller has a large avoidable performance gap under a feasible network condition.

For readability, define

$$R(e) := J(\pi_e^*; e) - J(\pi; e), \quad \hat{R}(e) := J(\hat{\pi}_e^*; e) - J(\pi; e). \quad (5)$$

Here $R(e)$ is the exact regret objective and $\hat{R}(e)$ is its practical approximation.

Main guarantee. Under realistic assumptions stated in Appendix A, let \tilde{e} be the scenario returned by the outer RL solver, let e^* maximize the exact objective in Eqn. (2), let ε denote the outer-solver suboptimality for Eqn. (4), and let $\delta(e)$ bound the inner-reference error in scenario e . Then

$$\max_{e \in \mathcal{E}} R(e) - R(\tilde{e}) \leq \varepsilon + \delta(e^*), \quad (6)$$

or equivalently,

$$R(\tilde{e}) \geq \max_{e \in \mathcal{E}} R(e) - \varepsilon - \delta(e^*). \quad (7)$$

Equation (7) is the key takeaway. The exact regret achieved by the returned scenario is itself a lower bound on the worst-case exact regret over the feasible search space. If the outer RL solver is near-optimal and the practical reference is accurate, then this lower bound is tight. The returned scenario is therefore not merely a hard example. It is a near-tight certificate of controller weakness in the searched regime.

Supporting results. The proof is built from two ingredients, both deferred to Appendix A. First, the outer RL solver returns an ε -optimal scenario for the approximate objective:

$$\hat{R}(\tilde{e}) \geq \max_{e \in \mathcal{E}} \hat{R}(e) - \varepsilon. \quad (8)$$

Second, the approximate regret is a pointwise lower bound on the exact regret, with gap controlled by the inner-reference error:

$$0 \leq R(e) - \hat{R}(e) \leq \delta(e), \quad \forall e \in \mathcal{E}. \quad (9)$$

Combining Eqns. (8) and (9) yields Theorem 1 below.

Theorem 1. *Under A1–A4, the scenario \tilde{e} returned by REGUARD certifies a lower bound on the worst-case exact regret objective, and this lower bound is within $\varepsilon + \delta(e^*)$ of the exact worst case (Eqns. 6 and 7).¹*

Why this guarantee matters. It directly quantifies the strength of the discovered failure case. The only sources of looseness are the outer-search error ε and the inner-reference error $\delta(e^*)$. As either component improves, the certificate strengthens immediately. This makes the formulation useful both analytically and operationally: it identifies a concrete failure scenario and quantifies how close that scenario is to the worst avoidable failure in the feasible regime under study.

4.3 Implementation

High-level design. Table 1 summarizes how REGUARD instantiates Eqn. (4) across three use cases detailed in Appendix B: congestion control, adaptive bitrate streaming, and load balancing. Each implementation exposes a small set of scenario variables to the outer RL solver, keeps the pretrained controller π fixed, and constructs a practical reference policy $\hat{\pi}_e^*$ under the same generated scenario. At time t , the solver observes the controller-induced system state, chooses the next resource element u_t , and appends it to the scenario $e = (u_0, \dots, u_{T-1})$. REGUARD then evaluates π and the reference under this shared scenario and uses the resulting regret from Eqn. (4) as the learning signal.

The main implementation contract is to make the comparison fair and diagnostic. REGUARD must keep the exogenous

¹Proof in Appendix A.6.

Use Case	RL Type	Training Environment	Network Scenario Variables	Reward (performance): $J(\pi; e)$	Reference Policy
CCA (Sage [40])	Offline	Emulation	Bandwidth, loss, delay	$\sum_t (\alpha \cdot \text{rate}_t + \beta \cdot \text{rtt}_t - \gamma \cdot \text{loss}_t)$	Best performance among Cubic, BBR, and NewReno.
ABR (Pensieve [26])	Online	Simulation	Bandwidth	$\sum_t (\alpha \cdot b_t - \beta \cdot r_t - \gamma \cdot b_t - b_{t-1})$	Exact K -step rolling bitrate oracle.
LB (Park [27])	Online	Simulation	Job arrival time, job size	$\alpha \cdot \int_0^T N_{\text{active}}(t) dt$	Best performance among LCT, JSQ, and CFS.

Table 1: Implementation summary for the instantiations of Eqn. (4) for the three use cases described in Appendix B. For Sage, $\alpha = 0.60$, $\beta = 0.25$, and $\gamma = 0.15$; rate_t is delivery rate divided by path capacity and clipped to $[0, 1]$, rtt_t is base RTT divided by current RTT and clipped to $[0, 1]$, and loss_t is loss normalized by path capacity and clipped to $[0, 1]$. For Pensieve, b_t is bitrate in Kbps, r_t is rebuffering time, $\alpha = \gamma = 10^{-3}$, and $\beta = 4.3$. For Park, $N_{\text{active}}(t)$ is the number of active jobs, T is the episode horizon, and $\alpha = 1/\text{reward_time_scale}$. LCT denotes Least Completion Time, JSQ denotes Join Shortest Queue, and CFS denotes Choose Fastest Server.

scenario shared, keep each policy’s closed-loop state isolated, and measure the reward gap using the application-specific reward in Table 1. The solver is therefore not rewarded for making every policy perform poorly. It is rewarded for finding challenging conditions where π performs poorly while $\hat{\pi}_e^*$ still obtains high reward. Each implementation defines $\hat{\pi}_e^*$ as the best policy in its practical reference class:

$$J(\hat{\pi}_e^*; e) := \underset{\pi' \in \Pi}{\text{maximize}} J(\pi'; e). \quad (10)$$

Eqn. (10) defines the reference used by the implementation over its chosen Π , even though this reference may still approximate the true scenario-specific optimum in Eqn. (3). For Sage [40] and Park [27], Π is a portfolio because prior work shows that different hand-designed policies perform best in different operational regimes, making their upper envelope a strong practical reference [4, 33, 40]. A *stronger* $\hat{\pi}_e^*$ improves the reward signal for the outer solver and reduces the oracle error $\delta(e)$ in A4, which directly tightens the certificate in Theorem 1.

Congestion control. The instantiation of Sage [40] (Fig. 3) is systems-heavy because REGUARD evaluates a real congestion-control stack inside process-level emulation rather than a pure simulator. REGUARD extends Mahimahi [32] from a passive trace player into an online-controlled, multi-policy emulator. At each outer step, u_t specifies the bottleneck link conditions for the next control interval. The common training mode maps a normalized solver action logarithmically to a shared bottleneck bandwidth, while richer configurations also control loss and propagation delay. The logarithmic bandwidth map is useful because transport behavior often changes by ratios rather than absolute Mbps increments. It gives the solver resolution in both low-capacity and high-capacity regimes without exposing an unnecessarily large action space.

Sage uses a best-of-portfolio reference over conventional congestion controllers. REGUARD runs Sage, BBR, Cubic, Reno, and any additional reference policies under the same link evolution, then applies Eqn. (10) using the normalized

transport reward in Table 1. The reward combines delivery rate relative to available capacity, RTT inflation relative to the base RTT, and loss normalized by capacity. This normalization makes scores comparable across the bandwidth range that the solver can generate. The objective in Eqn. (4) then favors selective performance failures: bandwidth transitions, latency regimes, or loss bursts where Sage underreacts, overreacts, or follows a poor recovery trajectory while at least one conventional controller remains effective.

The central Sage challenge is synchronized multi-policy emulation. REGUARD launches one isolated Mahimahi child instance per policy, with separate ports, actor identifiers, runtime directories, TCP state, queue dynamics, and shared-memory observation channels. Isolation prevents one controller’s packets, queues, or control history from contaminating another controller’s trajectory. At the same time, REGUARD treats these child instances as one logical experiment by applying the same u_t to every child at the same logical step and effective timestamp. This design turns the weak condition that the same action was sent into the stronger condition that the same link change was actually installed for every policy before the gap was scored.

The Mahimahi extension relies on an online control plane rather than a fixed trace loaded at startup. The control plane carries per-direction bandwidth, loss, delay, queue parameters, logical step identifiers, flags, and the absolute time at which an update should become effective. The emulator reports telemetry back to the solver, including the applied link settings, queue occupancy, queue delay, departure rate, dropped packets, dropped bytes, and applied logical step. REGUARD uses this telemetry to detect hidden divergence, such as stale link settings, skewed update times, placeholder observations, or child processes that exit early. When the synchronized comparison cannot be trusted, the step is truncated rather than silently producing a misleading gap.

The Sage implementation also supports structured short-timescale loss inside a solver interval. Instead of only applying

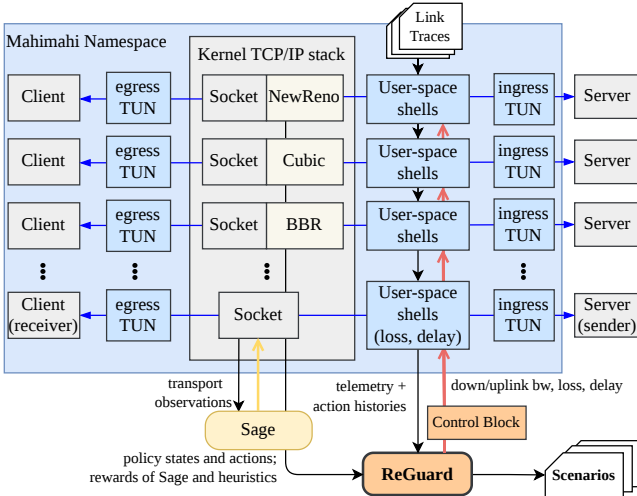


Figure 3: The key design point of Sage’s instantiation is end-to-end counterfactual consistency: REGUARD runs Sage and the reference controllers in synchronized but isolated emulation, then reuses the resulting controller-reference trajectories to identify risky states and derive small corrections.

an independent scalar loss probability, REGUARD can realize deterministic loss over short bins, which lets the solver express bursty or temporally correlated loss patterns while keeping u_t compact. This detail matters because many transport failures arise from temporal structure, not just average capacity or average loss. Together, online link control, lockstep emulation, and best-of-portfolio scoring make the Sage implementation a diagnostic testbed for policy-specific congestion-control failures.

Other controllers. Due to space constraints, the corresponding Pensieve and Park implementations appear in Appendix C.1.

5 Protecting RL Controllers against Performance Failures

REGUARD uses the scenarios found by the search in §4.3 to learn lightweight inference-time protection for the pretrained controller. This protection does not retrain the controller, replace it with a reference policy, or run the reference policy online. Instead, it distills controller-reference comparisons from challenging scenarios into interpretable rules that decide when to trust the controller and when to apply a small bounded correction. This turns failure discovery into protection: the search exposes where the controller leaves reward on the table, and REGUARD converts those failures into targeted interventions that preserve nominal behavior elsewhere.

5.1 Identifying Risky Control States

Failure discovery alone does not reveal which decisions caused the performance gap. A high-regret scenario shows that

the controller underperforms, but it does not say which states are risky, whether the controller is too aggressive or too conservative, or how the action should change. REGUARD addresses this gap with a *Counterfactual Analyzer* (Fig. 2), which treats the reference-policy portfolio as a set of counterfactual teachers. For each decision point t , it records the controller state s_t , the controller action $a_t = \pi(s_t)$, the reference action induced by $\hat{\pi}_e^*$, and the rewards achieved under the same scenario.

The Counterfactual Analyzer converts trajectory-level failures into state-level supervision. It identifies states where the controller is consistently suboptimal, compares the controller action with the reference action under the same observed state, labels the corrective direction, and summarizes recurring discrepancies as operator-meaningful patterns such as overly aggressive behavior, overly conservative behavior, or instability in a specific regime. A state is risky only when the controller leaves achievable reward on the table:

$$\text{Risky}(s_t) = 1 \iff J(\hat{\pi}_e^*; e) - J(\pi; e) \text{ is large.}$$

This criterion keeps REGUARD’s protection focused on avoidable controller mistakes rather than intrinsically hard scenarios where every policy performs poorly.

The same counterfactual comparison provides the corrective label. REGUARD uses a small application-independent label set:

$$\mathcal{C}_{\text{adjustment}} := \{\text{ABSTAIN}, \text{BACK_OFF}, \text{PUSH_HARDER}\}. \quad (11)$$

ABSTAIN leaves the controller action unchanged. BACK_OFF means the controller is acting too aggressively for the observed state. PUSH_HARDER means the controller is acting too conservatively. When the regret is small, the reference is not reliably better, or the corrective direction is ambiguous, the Counterfactual Analyzer outputs ABSTAIN. Thus, the heuristic portfolio is not merely a benchmark; it is the source of counterfactual action labels that make protection actionable.

5.2 Learning Protection Rules from Risky States

REGUARD learns its inference-time protection as threshold-based logic rules over observable controller state variables. The goal is not to fit an opaque surrogate for the controller. The goal is to learn hard associations between state regimes and the adjustment labels produced by the Counterfactual Analyzer. To do this, REGUARD uses NetNomos [17], a rule-learning framework that extracts compact logic rules from network data and predicates.

Predicate construction. REGUARD constructs the predicate vocabulary from normal controller behavior. It runs the controller on normal traces, computes percentile thresholds for each observable state variable, and converts those thresholds into predicates. For a raw feature $x_i(s)$, examples include

$$x_i(s) \geq q_{i,95}^{\text{normal}}, \quad x_i(s) \leq q_{i,25}^{\text{normal}}, \quad x_i(s) \geq q_{i,90}^{\text{normal}},$$

where $q_{i,p}^{\text{normal}}$ is the p th percentile of feature i on normal traces. These predicates make rules interpretable because each condition is expressed relative to the controller’s nominal operating regime. For example, they can capture unusually high loss or RTT inflation for Sage, low buffer or high download delay for Pensieve, and load imbalance or large incoming jobs for Park. **Rule synthesis.** NetNomos receives the predicates and the Counterfactual Analyzer’s adjustment labels. It learns implications of the form

$$\phi_1(s) \wedge \dots \wedge \phi_k(s) \implies \text{adjustment}(s) = c, \quad c \in C_{\text{adjustment}}.$$

The learned rules are deliberately simple: when their predicates match, they prescribe one adjustment. This keeps the deployed mechanism small, makes the learned rules inspectable, and gives the operator a concise explanation of when the controller tends to fail and what direction the correction should take.

Iterative refinement. The initial protection rules may still miss residual failure states that are slightly less severe than the ones initially found, so REGUARD refines them through a dataset-refinement loop. Each round trains the scenario search against the *currently protected* controller, not the original unprotected controller. The new scenarios are therefore counterexamples to the current protection rules: they expose states that bypass or stress the rules already learned. REGUARD replays these scenarios, uses the Counterfactual Analyzer to compute oracle gaps and per-action counterfactual labels for the newly visited states, and adds the resulting examples to a cumulative dataset.

REGUARD then re-learns the rule set *from scratch* over the aggregated dataset rather than appending patches to the old rules. This design is the key distinction. The dataset evolves across refinement rounds, but each rule set is synthesized as one coherent protection policy over all evidence collected so far. This approach avoids logical conflicts between old and new clauses, prevents rule explosion from accumulating special-case exceptions, and lets the learner revise earlier boundaries when later counterexamples show that they were too permissive or too restrictive.

The refinement loop focuses learning capacity where it matters most. If the current protection already covers one class of risky states, the next search is pushed toward remaining blind spots. Because earlier examples are retained, the learner also preserves evidence about where to intervene and where to abstain, reducing overfitting to the latest scenario distribution. As a result, refinement broadens coverage of truly risky regimes while keeping the final rule set compact, internally consistent, and grounded in observable decision-time features.

5.3 Efficient Inference-time Rule Enforcement

The learned protection operates at every controller decision point during inference [16], but enforcement is cheap because it is memoryless. At time t , it evaluates predicates on the current observable state s_t and adjusts only the controller’s proposed

action. It does not track full trajectories, run reference policies, estimate regret online, or solve an online optimization problem.

This design matches the standard RL interface. RL controllers already encode recent history, such as throughput samples, delay samples, prior actions, or queue summaries, into the current state. Under the Markov assumption [11, 19, 28, 30, 35], the current state captures the information needed for the next decision. REGUARD uses the same premise for protection: if a risky regime appears in s_t , a rule over s_t is sufficient to protect the action.

Formally, let \mathcal{R} be the learned rule set and let $\pi(s_t)$ be the action proposed by the pretrained controller. REGUARD computes

$$c_t := g_{\mathcal{R}}(s_t) \quad \text{and} \quad a_t^{\text{prot}} := h(\pi(s_t), c_t, s_t),$$

where $g_{\mathcal{R}}$ maps the current state to an adjustment label and h maps that label to an application-specific correction. If $c_t = \text{ABSTAIN}$, then $a_t^{\text{prot}} := \pi(s_t)$. If c_t is `BACK_OFF` or `PUSH_HARDER`, then h applies a bounded correction in the direction supported by the Counterfactual Analyzer’s labels. The controller remains the primary decision maker, since REGUARD intervenes only when the learned rules provide a clear reason to do so.

REGUARD further reduces enforcement cost with a prefix trie [6]. Each rule is a conjunction of boolean predicates, so rules with shared predicate prefixes can be stored in a common tree. At runtime, REGUARD evaluates predicates along reachable trie paths and prunes entire subtrees as soon as a required predicate is false. This method avoids scanning every rule independently and makes enforcement depend on the matched predicate structure rather than the raw number of rules. As a result, the controllers protected by REGUARD meet their inference-time deadlines in our experiments (Fig. 7).

5.4 Implementation

Congestion control. For Sage [40], the protected controller consumes the 69-dimensional policy observation summarized in Table 2. These features cover transport signals such as RTT, RTT variation, delivery rate, recent delivery-rate summaries, loss, min-RTT ratios, prior action values, elapsed time, and derived congestion or utilization ratios. Sage’s action space is continuous, so the generic adjustments are implemented as bounded nudges to the primary scalar control dimension. If REGUARD abstains, Sage’s action is left unchanged: $a_t^{\text{prot}} := a_t$. If REGUARD deduces `BACK_OFF`, it decreases the scalar action by a small step Δ and clips the result to the legal action range: $a_t^{\text{prot}} := \text{clip}(a_t - \Delta, a_{\min}, a_{\max})$. If REGUARD deduces `PUSH_HARDER`, it increases the action symmetrically: $a_t^{\text{prot}} := \text{clip}(a_t + \Delta, a_{\min}, a_{\max})$. This implementation preserves Sage as the base controller and changes only the degree of aggressiveness when the rules identify a state where the Counterfactual Analyzer indicates a clear corrective direction.

Other controllers. Pensieve and Park instantiate the same architecture with application-specific state predicates and bounded action corrections. In all cases, the Counterfactual Analyzer provides the risky-state labels and corrective directions, NetNomos turns them into interpretable rules, and REGUARD applies only small adjustments when those rules fire. Due to space constraints, the detailed Pensieve and Park protection implementations appear in Appendix C.2.

6 Evaluation

We aim to answer the following evaluation questions through experiments:

- E1. Can REGUARD find network scenarios where a given RL controller incurs a larger performance gap than competing baselines expose?
- E2. Can REGUARD effectively protect a given RL controller without retraining while preserving its nominal performance?
- E3. How does the hardness of the source counterfactual affect the effectiveness of REGUARD’s protection?
- E4. How much overhead does REGUARD impose on the RL controller during inference?
- E5. Can REGUARD iteratively reduce the performance gap of the most challenging network scenarios to the level of normal scenarios?

6.1 Setup

Network Traces. We leverage *diverse network traces*² to ensure that the RL controllers experience a wide range of network scenarios during training:

- Sage [40] is trained and evaluated on intra-continental live Internet paths using servers across 16 U.S. cities, intercontinental live Internet paths using 13 servers outside the U.S., and highly variable cellular workloads from 23 cellular traces gathered in NYC. We use these original traces and follow the same train-test split [3, 40].
- Pensieve [26] is trained and evaluated on 375 FCC broadband traces [1] and 425 Norway cellular traces [2]. We follow the train-test split specified in the Genet paper [38].
- Park [27] is trained and evaluated on 200 traces of the 5 workload types used in prior work [38], namely, “Default”, “Original”, “RL1”, “RL2”, and “RL3.” These scheduling workloads differ substantially in service rate, job size, job interval, number of jobs, and queue-shuffled probability. We use half of the traces for training and the other half for testing.

Baselines for discovering challenging network scenarios.

- **Indago** [8] learns a surrogate failure predictor from the RL agent’s training episodes, then uses that predictor’s output as a fitness function and its saliency gradients as guidance

to search for environment scenarios that are likely to make the RL agent fail.

- **Genet** [38] is designed to help RL controllers learn to deal with challenging network scenarios by generating a challenging learning curriculum that maximizes the raw gap between the RL controller and a single rule-based baseline policy, aka “gap-to-baseline.” For these baselines used by Genet, we use BBR [10] for CCA, RobustMPC [41] for ABR, and least-load-first for LB.

- **Gilad et al.** [13] directly searches for challenging network conditions that degrade the controller’s performance.

- **Random** explores the space of possible network scenarios randomly by uniformly fuzzing the entire domains of network scenario variables (Table 1).

Protection baselines. To protect the RL controller against challenging network scenarios, we compare REGUARD³ against the following retraining-based approaches:

- **Fine-tuning:** The most direct way to improve an RL policy on a given workload family is to continue training it on that workload. We consider two fine-tuning settings: **(individual)** We fine-tune the RL controller on traces from discovered scenarios in one specific family, and evaluate the resulting policy on held-out test traces of the *same* kind (*e.g.*, fine-tuning on Indago’s traces and then testing on Indago’s held-out traces). **(all)** We fine-tune the RL controller on traces from *all* scenario families (namely, REGUARD, Indago, Genet, Gilad et al., and Random) and evaluate the resulting policy separately on held-out traces from each individual family.

- **Curriculum learning:** We use Genet [38] to iteratively generate increasingly challenging network scenarios and continue training the RL policy on the discovered traces, thereby exposing it to a progressively more challenging curriculum [5, 15]. For all use cases, we run Genet for 20 rounds, doubling the 10 rounds used in the original paper.

Note that, since Sage [40] is an offline RL method, to fine-tune it we follow the same procedure used by Mowgli [4]: we first collect trajectories of the RL policy on the discovered challenging network scenarios and then continue training the model offline on those trajectories. Testbed setup is described in Appendix G.

6.2 Results

Finding 1: REGUARD finds the largest performance gaps in all three use cases, certifying severe performance gaps of RL controllers.

Results for E1 are clear. Fig. 4 shows that REGUARD finds the most challenging scenarios in all three applications, and the separation from prior baselines is large rather than marginal.

³The logic rules are always derived from training traces, never from held-out test traces.

²These traces are referred to as “Normal” in the following experiments.

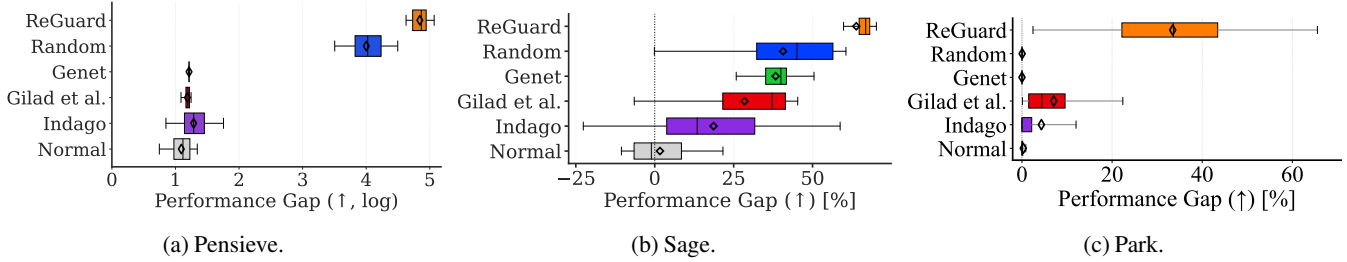


Figure 4: REGUARD finds the most challenging scenarios in all three use cases. The gap is consistently larger than the strongest baseline and far above the Normal regime, showing that REGUARD exposes large avoidable underperformance rather than marginally harder tests.

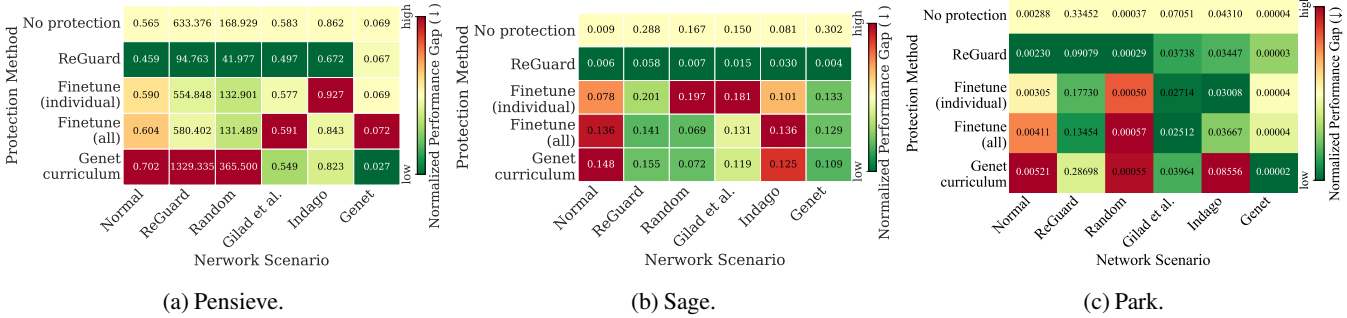


Figure 5: REGUARD provides the strongest overall protection when derived from REGUARD scenarios, while preserving nominal performance. The results of REGUARD shown here come from its *very first iteration*, and later iterations provide even more protection (see Fig. 8). REGUARD is most effective on the most challenging scenarios, but it also transfers to other, less challenging scenario families, indicating that it captures recurring risky states rather than memorizing patterns from one method.

For Pensieve, we report the log performance gap because the underlying QoE gap can explode under long rebuffering. For Sage and Park, we report the fraction of achievable performance that the controller fails to realize. Under either metric, REGUARD remains clearly above every baseline. The strongest separations are about $7\times$ larger than the next-best baseline in Pensieve, $1.6\times$ larger in Sage, and $6.2\times$ larger in Park. This answers E1: REGUARD is the only method that consistently exposes large avoidable underperformance instead of slightly harder tests.

Finding 2: REGUARD, when derived from the largest-gap network scenarios it finds, provides the strongest protection among the compared methods in all three use cases, preserves nominal performance, and remains effective beyond its source scenario family.

On E2, Fig. 5 shows that REGUARD, when derived from its own scenarios, gives the strongest overall protection without retraining. We report normalized performance gap, which asks how much avoidable underperformance remains after accounting for the scale of each scenario, so lower is better. On the most challenging family for each application, REGUARD removes about 80% of the gap. The same protection also transfers to other scenario families,

which shows that REGUARD is capturing recurring risky states rather than memorizing one generator. The Normal column makes the practical tradeoff clear. REGUARD preserves or improves nominal performance, whereas fine-tuning and curriculum learning often degrade it. Appendix E reports the detailed reductions and per-family comparisons.

Finding 3: The more challenging the source counterfactual is, the more protection REGUARD offers.

Regarding E3, Fig. 6 isolates why the previous result holds. We report average protection percentage, *i.e.*, how much of the original gap REGUARD removes on average, so higher is better. Across all three applications, REGUARD is consistently stronger when it is derived from more challenging counterfactuals. This means the search stage is not just producing examples. It is producing the high-gap states that make the rule learner effective.

Finding 4: REGUARD always fits within the inference time budget.

On E4, Fig. 7 shows that REGUARD stays well within the on-line decision budget in every system. We report decision-time ratio, the fraction of the available inference budget consumed by the controller plus protection, so any value below 100%

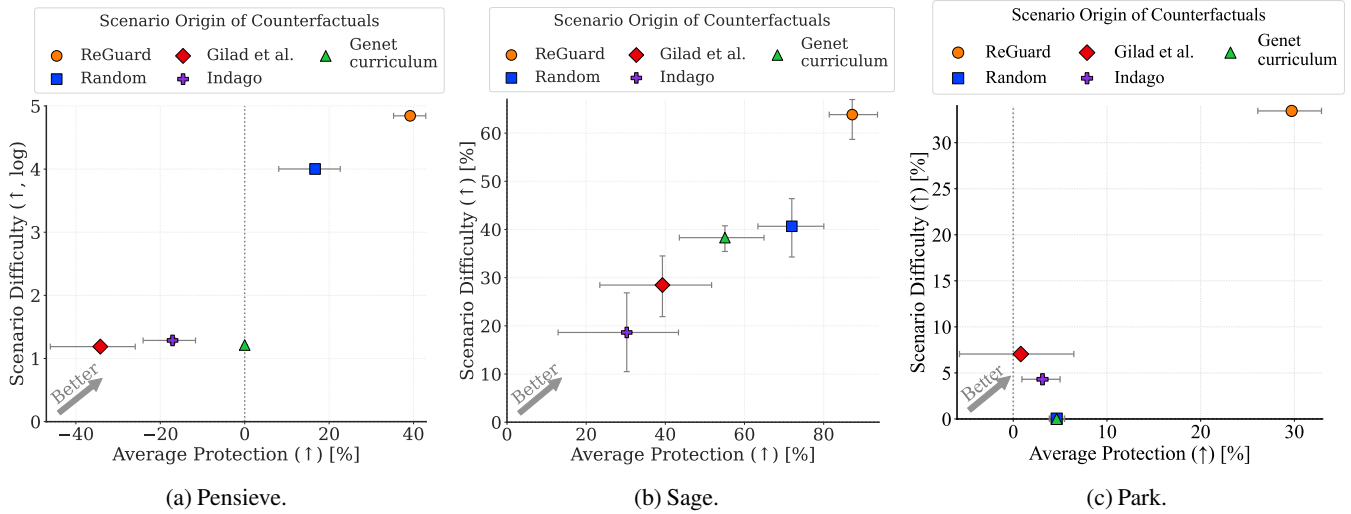


Figure 6: Harder counterfactual sources yield stronger protection from REGUARD. Search quality therefore matters directly: better discovery produces better protection.

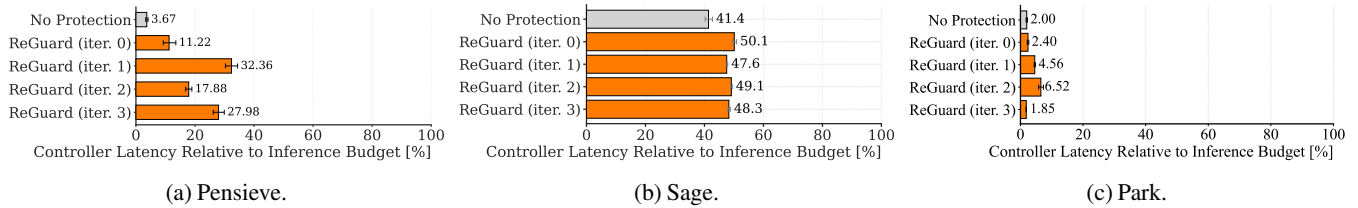


Figure 7: REGUARD remains well within the online decision budget in all three systems across all refinement iterations. Later iterations strengthen protection, but they do not introduce systematic latency growth.

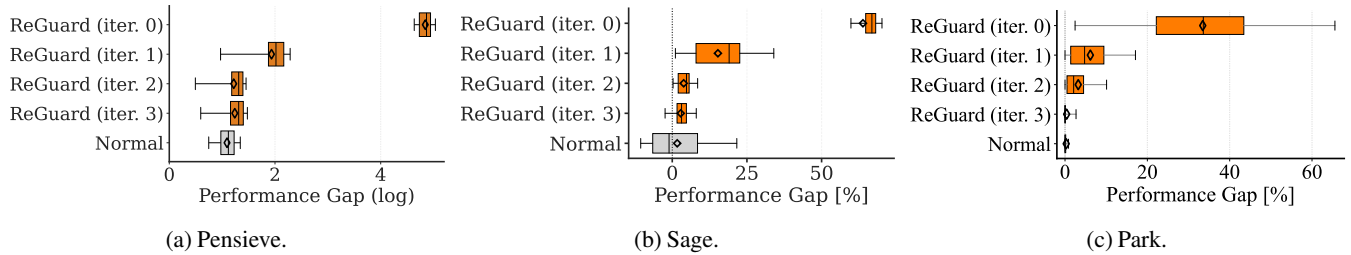


Figure 8: A few search-and-protect iterations are enough to drive the discovered gap close to the Normal regime. Most of the improvement arrives in the first two rounds, after which the remaining challenging cases are much less severe.

meets deadline. Pensieve must decide before the current chunk finishes downloading, Sage has a fixed 10ms interval, and Park must place the current job before the next arrival. Across iterations, REGUARD uses 11.22–32.36% of Pensieve’s budget, 47.55–50.11% of Sage’s budget, and 1.85–6.52% of Park’s budget. Later iterations provide stronger protection, but the runtime overhead does not grow with that improvement. This answers E4: REGUARD is deployment-feasible.

Finding 5: REGUARD does not merely protect against a

specific challenging scenario; it fundamentally shrinks the worst-case performance gap.

Finally, Fig. 8 answers E5. It uses the same performance-gap metrics as Fig. 4, so lower and closer to the Normal line is better. Across all three applications, most of the improvement arrives in the first two search-and-protect rounds. By iteration 3, the mean discovered gap falls by roughly 75% in Pensieve, 94% in Sage, and 93% in Park, and the remaining challenging cases are close to the Normal regime. Detailed trajectories appear in Appendix E.

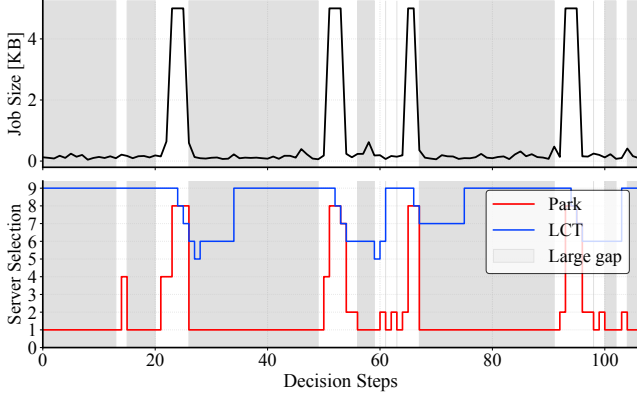


Figure 9: Small-job-heavy scenarios expose Park’s bias toward slow servers. Park keeps routing tiny jobs to the slow tier even while fast servers remain idle, which creates prolonged queuing and a large gap to Least Completion Time (LCT).

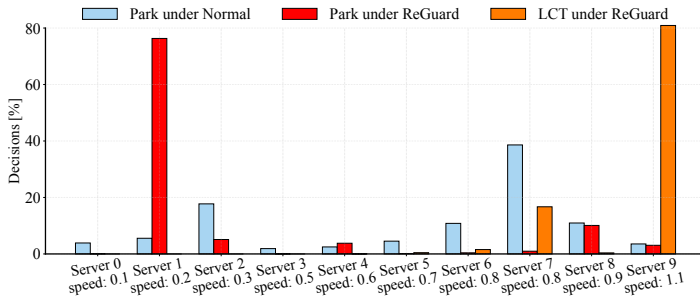


Figure 10: Network scenarios found by REGUARD turn Park’s mild preference under nominal conditions into a near-collapse onto slow Server 1. LCT still routes most jobs to the fast tier, which shows that the issue is Park’s dispatch policy rather than an inherently bad workload.

7 Analyzing Revealed Performance Failures

The challenging scenarios discovered by REGUARD do not create arbitrary worst cases. In Park, they expose a specific structural weakness that also appears in REGUARD’s logic rules: when small jobs dominate the workload, Park collapses onto slow servers even though fast servers remain available. Due to space constraints, the main paper keeps only the core Park mechanism here. Detailed supporting statistics for Park appear in Appendix F.1, while the corresponding Pensieve and Sage analyses appear in Appendix F.2 and Appendix F.3. Below, pk denotes the corresponding k th-percentile predicate computed from normal scenarios (§5).

Park’s failure is a structural dispatch bias rather than simple overload. Fig. 9 shows a slice dominated by small jobs that the fast tier can easily absorb, yet Park still collapses onto the slow tier. For jobs of at most 0.2KB, Park chooses a slow server 97.3% of the time, whereas Least Completion Time routes 81.1% of those same jobs to the fast tier. Even inside

the large-gap region, fast servers remain idle throughout. The problem is therefore not a lack of capacity. It is that Park systematically prefers the wrong part of the cluster.

Fig. 10 shows that this is not a one-off slice. Under the challenging scenarios discovered by REGUARD, Park places 76.3% of all decisions on server 1, up from 5.6% on normal workloads. Least Completion Time under the same scenarios still routes most decisions to the fast tier. At least one fast server remains idle on 98.0% of steps, so the issue is not universal overload. The challenging scenario amplifies a policy bias that sends small jobs to the wrong part of the cluster.

REGUARD makes this failure explicit by separating risk detection from recovery. Here, GapToMin_i denotes server i ’s load proxy minus the current minimum load proxy, and PostLoad_i denotes server i ’s load proxy after hypothetically adding the incoming job. One representative risk rule is

$$(\text{JobSize} \leq p25) \wedge (\text{GapToMin}_1 > p25) \\ \wedge (\text{PostLoad}_9 \leq p25) \wedge (\text{StdLoad} > p25) \implies \text{Risky} = \text{true}.$$

This rule says that a small job has arrived, server 1 already looks much busier than the currently least-loaded server, server 9 would still look light after taking the job, and the cluster is already imbalanced enough for that difference to matter. That is exactly the failure state in Fig. 9: the next job is easy for the fast tier, but Park is still feeding the slow tier.

Once such a state is marked risky, the recovery rules reopen the fast servers explicitly. A representative correction rule is

$$(\text{GapToMin}_1 > p25) \wedge (\text{GapToMin}_9 \leq p25) \\ \wedge (\text{MeanLoad} \leq p25) \wedge (\text{TotalLoad} \leq p25) \implies \text{Allow}_9 = \text{true}.$$

This rule says that when server 1 is far from the current minimum but server 9 remains near that minimum under low overall load, the fastest server must remain available to the policy. The rule does not talk about a mysterious latent bias. It states directly that Park should stop treating server 1 as attractive when server 9 is visibly better. Together, these rules show that REGUARD is not merely labeling workloads as challenging. It identifies the specific state in which Park’s dispatch preference becomes harmful and reopens the fast tier only in that state. Additional supporting statistics and correction rules appear in Appendix F.1.

8 Conclusion

REGUARD discovers worst-case failures in RL-based network controllers by formulating the search for challenging network scenarios as bilevel regret maximization. It protects the controller at inference time using lightweight, interpretable logic rules derived from counterfactual analysis. Across three controllers spanning distinct networking tasks, REGUARD exposes performance gaps up to $6\times$ larger than prior methods and closes up to 85% of those gaps within the first refinement iteration, without retraining and without sacrificing nominal performance.

References

- [1] Measuring broadband america. <https://www.fcc.gov/general/measuring-broadband-america>.
- [2] Dataset: Hsdpa-bandwidth logs for mobile http streaming scenarios. <https://skulddata.cs.umass.edu/traces/mmsys/2013/pathbandwidth/>.
- [3] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 632–647, 2020.
- [4] Neil Agarwal, Rui Pan, Francis Y. Yan, and Ravi Netravali. Mowgli: Passively learned rate control for real-time video. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 579–594, 2025. URL <https://www.usenix.org/conference/nsdi25/presentation/agarwal>.
- [5] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociey, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik’s cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.
- [6] Ryan Beckett, Francis Y. Yan, Raghunadha Reddy Pocha, Vineesh V. Raj, Ayyub Shaik, and Siva Kesava Reddy Kakarla. Concord: Learning network configuration contracts. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys ’26)*, page 18, Edinburgh, Scotland, UK, April 2026. ACM. ISBN 979-8-4007-2212-7/26/04. doi: 10.1145/3767295.3769338.
- [7] Matan Ben-Tov, Daniel Deutch, Nave Frost, and Mahmood Sharif. Cafa: Cost-aware, feasible attacks with database constraints against neural tabular classifiers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1345–1364. IEEE, 2024.
- [8] Matteo Biagiola and Paolo Tonella. Testing of deep reinforcement learning agents with surrogate models. *ACM Transactions on Software Engineering and Methodology*, 33(3):73:1–73:33, 2024. doi: 10.1145/3631970.
- [9] Stephen Boyd and Lieven Vandenbergh. *Convex optimization*. Cambridge university press, 2004.
- [10] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.
- [11] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.
- [12] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. Verifying learning-augmented systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 305–318, 2021.
- [13] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 85–92. ACM. ISBN 978-1-4503-7020-2. doi: 10.1145/3365609.3365862. URL <https://dl.acm.org/doi/10.1145/3365609.3365862>.
- [14] Allen A Goldstein. Convex programming in hilbert space. *Bulletin of the American Mathematical Society*, 70(5):709–710, 1964.
- [15] Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pages 1311–1320. Pmlr, 2017.
- [16] Hongyu Hè and Maria Apostolaki. Just-in-time logic enforcement: A new paradigm of combining statistical and symbolic reasoning for network management. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets 25)*, pages 184–192, 2025.
- [17] Hongyu Hè, Minhao Jin, and Maria Apostolaki. Making Logic a First-Class Citizen in Network Data Generation with ML. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, 2026.
- [18] Arthur S. Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. AI/ML for Network Security: The Emperor has no Clothes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS ’22)*, Los Angeles, CA, USA, 2022. ACM. doi: 10.1145/3548606.3560609. URL <https://trusteeml.github.io/>.
- [19] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning (ICML)*, pages 3050–3059. PMLR, 2019.
- [20] Minhao Jin and Maria Apostolaki. Robustifying ml-powered network classifiers with pants. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

- [21] Ananya Kumar, Aditi Raghunathan, Robbie Matthew Jones, Tengyu Ma, and Percy Liang. Fine-tuning can distort pretrained features and underperform out-of-distribution. In *International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=UYneFzXSJWh>.
- [22] Aviral Kumar, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. Stabilizing off-policy q-learning via bootstrapping error reduction. *Advances in neural information processing systems*, 32, 2019.
- [23] Evgenii S Levitin and Boris T Polyak. Constrained minimization methods. *USSR Computational Mathematics and Mathematical Physics*, 6(5):1–50, 1966.
- [24] Qin-Wen Luo, Ming-Kun Xie, Ye-Wen Wang, and Sheng-Jun Huang. Optimistic critic reconstruction and constrained fine-tuning for general offline-to-online rl. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-3435.
- [25] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net>.
- [26] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM. ISBN 978-1-4503-4653-5. doi: 10.1145/3098822.3098843. URL <https://dl.acm.org/doi/10.1145/3098822.3098843>.
- [27] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems*, 32, 2019.
- [28] Andrew Kachites McCallum. *Reinforcement learning with hidden states*. PhD thesis, University of Rochester, 1995.
- [29] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 154–171. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi: 10.1145/3387514.3405859. URL <https://dl.acm.org/doi/10.1145/3387514.3405859>.
- [30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
- [31] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. Finding adversarial inputs for heuristics using multi-level optimization. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 927–949, 2024.
- [32] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.
- [33] Lorenzo Pappone, Alessio Sacco, and Flavio Esposito. Mutant: Learning congestion control from existing protocols via online reinforcement learning. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1507–1522, 2025.
- [34] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349, 2020. doi: 10.1109/SP40000.2020.00073.
- [35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [36] Brian Thompson, Jeremy Gwinnup, Huda Khayrallah, Kevin Duh, and Philipp Koehn. Overcoming catastrophic forgetting during domain adaptation of neural machine translation. In Jill Burstein, Christy Doran, and Tamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2062–2068, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1209. URL <https://aclanthology.org/N19-1209/>.
- [37] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=SyxAb30cY7>.
- [38] Zhengxu Xia, Yajie Zhou, Francis Y Yan, and Junchen Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, 2022.

- [39] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.
- [40] Chen-Yu Yen, Soheil Abbasloo, and H. Jonathan Chao. Computers can learn from the heuristic designs and master internet congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 255–274, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi: 10.1145/3603269.3604838. URL <https://doi.org/10.1145/3603269.3604838>.
- [41] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787486. URL <https://dl.acm.org/doi/10.1145/2785956.2787486>.

A Proofs for Regret Guarantees

This appendix gives the detailed assumptions, intermediate lemmas, and full proofs for the regret guarantees in Section 4.2. Our goal is to make explicit what is guaranteed by the formulation, what approximation errors enter the final bound, and how those errors propagate through the argument.

A.1 Setup and Notation

We begin by restating the exact and approximate regret objectives used in the main text. For every feasible network scenario $e \in \mathcal{E}$, define

$$R(e) := J(\pi_e^*; e) - J(\pi; e), \quad \hat{R}(e) := J(\hat{\pi}_e^*; e) - J(\pi; e). \quad (12)$$

Here:

- $R(e)$ is the *exact* regret objective, where the comparator π_e^* is an exact optimizer within the reference policy class:

$$\pi_e^* \in \operatorname{argmax}_{\pi \in \Pi} J(\pi; e).$$

- $\hat{R}(e)$ is the *approximate* regret objective, where the comparator $\hat{\pi}_e^* \in \Pi$ is the practical reference policy used by the implementation.

The exact bilevel target in the main text is

$$e^* \in \operatorname{argmax}_{e \in \mathcal{E}} R(e), \quad (13)$$

while the implemented procedure approximately solves

$$\hat{e} \in \operatorname{argmax}_{e \in \mathcal{E}} \hat{R}(e). \quad (14)$$

The proofs below establish three facts. First, the outer solver returns a near-optimal scenario for the approximate objective. Second, the approximate objective is a pointwise lower bound on the exact objective. Third, combining these two facts yields the final exact-regret guarantee in Theorem 1.

A.2 Assumptions

We restate the assumptions from the main text for completeness.

A1: The feasible set \mathcal{E} is non-empty, and both Eqns. (2) and (4) admit optimizers.

A2: For every $e \in \mathcal{E}$, the rewards $J(\pi_e^*; e)$, $J(\hat{\pi}_e^*; e)$, and $J(\pi; e)$ are finite.

A3: The outer RL solver returns an ε -optimal solution $\tilde{e} \in \mathcal{E}$ to Eqn. (4), namely

$$\max_{e \in \mathcal{E}} \hat{R}(e) - \hat{R}(\tilde{e}) \leq \varepsilon. \quad (15)$$

A4: The approximate reference satisfies

$$J(\pi_e^*;e) - J(\hat{\pi}_e^*;e) \leq \delta(e) \quad (16)$$

for every $e \in \mathcal{E}$, where $\delta(e) \geq 0$.

These assumptions are mild in the settings studied in the paper. **A1** says that the operator-defined feasible family of network scenarios is nontrivial and that both optimization problems are well posed. **A2** excludes degenerate cases in which the trajectory reward is undefined or unbounded. **A3** captures the quality of the outer solver. It does not require exact optimization, only that the scenario returned by the solver be within ε of the best achievable approximate-regret score. **A4** captures the quality of the inner reference policy. It says that the approximate reference does not fall more than $\delta(e)$ below the exact best reference in scenario e .

The final theorem separates these two error sources cleanly. The term ε accounts for suboptimality in the outer search over scenarios. The term $\delta(e^*)$ accounts for the error of the practical reference at an exact worst-case scenario. This separation is useful because the two terms arise from different components of the system and can be improved independently.

A.3 Auxiliary Observations

We first record two elementary but useful observations that will be used repeatedly in the proofs.

Lemma 2. For every feasible scenario $e \in \mathcal{E}$,

$$J(\pi_e^*;e) \geq J(\hat{\pi}_e^*;e). \quad (17)$$

Proof. Fix any $e \in \mathcal{E}$. By definition, π_e^* is an optimizer of $J(\pi';e)$ over the reference class Π . Since $\hat{\pi}_e^* \in \Pi$, the optimality of π_e^* implies

$$J(\pi_e^*;e) \geq J(\hat{\pi}_e^*;e).$$

This proves the claim. \square

Lemma 3. For every feasible scenario $e \in \mathcal{E}$,

$$R(e) - \hat{R}(e) = J(\pi_e^*;e) - J(\hat{\pi}_e^*;e). \quad (18)$$

Proof. By the definitions in Eqn. (12),

$$R(e) = J(\pi_e^*;e) - J(\pi;e), \quad \hat{R}(e) = J(\hat{\pi}_e^*;e) - J(\pi;e).$$

Subtracting the second identity from the first cancels the common term $J(\pi;e)$ and yields

$$R(e) - \hat{R}(e) = J(\pi_e^*;e) - J(\hat{\pi}_e^*;e). \quad \square$$

The next observation makes explicit that the approximate objective is always conservative.

Lemma 4. For every feasible scenario $e \in \mathcal{E}$,

$$\hat{R}(e) \leq R(e). \quad (19)$$

Proof. Fix any $e \in \mathcal{E}$. By Lemma 2,

$$J(\pi_e^*;e) \geq J(\hat{\pi}_e^*;e).$$

Subtracting the common term $J(\pi;e)$ from both sides preserves the inequality, giving

$$J(\pi_e^*;e) - J(\pi;e) \geq J(\hat{\pi}_e^*;e) - J(\pi;e).$$

By Eqn. (12), this is exactly

$$R(e) \geq \hat{R}(e). \quad \square$$

Lemma 4 is conceptually important. It says that replacing the exact reference policy with the practical reference can only decrease the regret score. In other words, the approximate regret objective used by the implementation is a pointwise lower bound on the ideal objective of interest.

A.4 Outer-search Guarantee

We now formalize the guarantee provided by the outer RL solver on the approximate objective.

Lemma 5. Under **A1** and **A3**, the scenario \tilde{e} returned by the outer solver satisfies

$$\max_{e \in \mathcal{E}} \hat{R}(e) - \hat{R}(\tilde{e}) \leq \varepsilon. \quad (20)$$

Equivalently,

$$\hat{R}(\tilde{e}) \geq \max_{e \in \mathcal{E}} \hat{R}(e) - \varepsilon. \quad (21)$$

Proof. By **A1**, the approximate outer problem in Eqn. (4) admits an optimizer. Let $\hat{e}^* \in \mathcal{E}$ be such that

$$\hat{R}(\hat{e}^*) = \max_{e \in \mathcal{E}} \hat{R}(e). \quad (22)$$

By **A3**, the outer solver returns a scenario $\tilde{e} \in \mathcal{E}$ whose approximate objective value is within ε of the optimum:

$$\max_{e \in \mathcal{E}} \hat{R}(e) - \hat{R}(\tilde{e}) \leq \varepsilon. \quad (23)$$

This is exactly Eqn. (20). Rearranging Eqn. (20) gives

$$\hat{R}(\tilde{e}) \geq \max_{e \in \mathcal{E}} \hat{R}(e) - \varepsilon,$$

which is Eqn. (21). \square

Although straightforward, Lemma 5 plays an important role. It tells us that the outer solver returns a scenario whose approximate regret score is nearly the largest possible over the feasible family. This is the first half of the final argument. The second half is to connect approximate regret to exact regret.

A.5 Pointwise Approximation Guarantee

We next show that the approximate regret is a pointwise lower bound on exact regret and that the gap is controlled exactly by the inner-reference error.

Lemma 6. *Under A2 and A4, every $e \in \mathcal{E}$ satisfies*

$$0 \leq R(e) - \hat{R}(e) \leq \delta(e). \quad (24)$$

Proof. Fix any $e \in \mathcal{E}$. We prove the lower and upper bounds separately.

Lower bound. By Lemma 3,

$$R(e) - \hat{R}(e) = J(\pi_e^*; e) - J(\hat{\pi}_e^*; e).$$

By Lemma 2,

$$J(\pi_e^*; e) - J(\hat{\pi}_e^*; e) \geq 0.$$

Combining the two equations yields

$$R(e) - \hat{R}(e) \geq 0. \quad (25)$$

Upper bound. Again by Lemma 3,

$$R(e) - \hat{R}(e) = J(\pi_e^*; e) - J(\hat{\pi}_e^*; e).$$

A4 states exactly that

$$J(\pi_e^*; e) - J(\hat{\pi}_e^*; e) \leq \delta(e).$$

Therefore,

$$R(e) - \hat{R}(e) \leq \delta(e). \quad (26)$$

Combining Eqns. (25) and (26) proves

$$0 \leq R(e) - \hat{R}(e) \leq \delta(e). \quad \square$$

Lemma 6 is the precise statement that the approximate objective is conservative. At every scenario e , the exact regret is at least as large as the approximate regret. Furthermore, the amount by which the approximate objective can underestimate the exact objective is no larger than the inner-reference error $\delta(e)$. This pointwise control is what allows us to lift an approximate-optimality guarantee for the outer solver into an exact-regret certificate.

A.6 Exact-regret Guarantee

We now prove the main guarantee from the paper.

Proof of Theorem 1. By A1, the exact outer problem in Eqn. (2) admits an optimizer. Let

$$e^* \in \operatorname{argmax}_{e \in \mathcal{E}} R(e). \quad (27)$$

Then by definition,

$$R(e^*) = \max_{e \in \mathcal{E}} R(e). \quad (28)$$

Our goal is to lower-bound $R(\tilde{e})$ in terms of $R(e^*)$ and then substitute Eqn. (28).

Step 1: compare the returned scenario to the best approximate-regret scenario. By Lemma 5,

$$\hat{R}(\tilde{e}) \geq \max_{e \in \mathcal{E}} \hat{R}(e) - \varepsilon. \quad (29)$$

Since $e^* \in \mathcal{E}$, the maximum over \mathcal{E} is at least the value at e^* . Therefore,

$$\hat{R}(\tilde{e}) \geq \hat{R}(e^*) - \varepsilon. \quad (30)$$

Step 2: relate exact and approximate regret at the returned scenario. Applying Lemma 6 to the scenario \tilde{e} gives

$$R(\tilde{e}) \geq \hat{R}(\tilde{e}). \quad (31)$$

Step 3: relate exact and approximate regret at the exact worst-case scenario. Applying Lemma 6 to the scenario e^* gives

$$0 \leq R(e^*) - \hat{R}(e^*) \leq \delta(e^*).$$

Rearranging the upper bound yields

$$\hat{R}(e^*) \geq R(e^*) - \delta(e^*). \quad (32)$$

Step 4: chain the inequalities. Starting from Eqn. (31) and then using Eqns. (30) and (32), we obtain

$$R(\tilde{e}) \geq \hat{R}(\tilde{e}) \quad (33)$$

$$\geq \hat{R}(e^*) - \varepsilon \quad (34)$$

$$\geq R(e^*) - \delta(e^*) - \varepsilon. \quad (35)$$

Substituting Eqn. (28) into Eqn. (35) gives

$$R(\tilde{e}) \geq \max_{e \in \mathcal{E}} R(e) - \varepsilon - \delta(e^*). \quad (36)$$

This is exactly Eqn. (7) in the main text.

Finally, rearranging Eqn. (36) yields

$$\max_{e \in \mathcal{E}} R(e) - R(\tilde{e}) \leq \varepsilon + \delta(e^*), \quad (37)$$

which is Eqn. (6) in the main text. This completes the proof. \square

A.7 Interpretation of the Final Bound

Theorem 1 has a clean interpretation.

First, the regret achieved by the returned scenario, $R(\tilde{e})$, is itself a *certificate value*. It lower-bounds the worst-case exact regret over the feasible family up to the additive error $\varepsilon + \delta(e^*)$. Thus, if $R(\tilde{e})$ is large, then the controller provably has a substantial avoidable performance gap in the searched regime.

Second, the theorem isolates the only two sources of looseness in the certificate:

- ε is the suboptimality of the outer search over scenarios. It is controlled by the quality of the RL-based outer solver.

- $\delta(e^*)$ is the suboptimality of the practical reference at an exact worst-case scenario. It is controlled by the quality of the inner reference policy.

Third, the dependence on $\delta(e^*)$ rather than a worst-case uniform bound over all $e \in \mathcal{E}$ is meaningful. The theorem only needs the inner-reference quality at an exact optimizer of the true objective. If the practical reference is especially accurate near the most informative failure scenarios, then the final certificate can still be tight even if the reference is less accurate elsewhere.

A.8 A Uniform-error Corollary

In some settings, it is convenient to summarize inner-reference quality by a uniform bound. The next corollary makes this explicit.

Corollary 7. *Suppose the assumptions of Theorem 1 hold, and in addition there exists a constant $\bar{\delta} \geq 0$ such that*

$$\delta(e) \leq \bar{\delta}, \quad \forall e \in \mathcal{E}. \quad (38)$$

Then the returned scenario \tilde{e} satisfies

$$\max_{e \in \mathcal{E}} R(e) - R(\tilde{e}) \leq \varepsilon + \bar{\delta}, \quad (39)$$

or equivalently,

$$R(\tilde{e}) \geq \max_{e \in \mathcal{E}} R(e) - \varepsilon - \bar{\delta}. \quad (40)$$

Proof. By Theorem 1,

$$\max_{e \in \mathcal{E}} R(e) - R(\tilde{e}) \leq \varepsilon + \delta(e^*).$$

Under Eqn. (38), we have $\delta(e^*) \leq \bar{\delta}$. Substituting this into the previous inequality yields Eqn. (39). Rearranging gives Eqn. (40). \square

Corollary 7 is weaker than Theorem 1, but sometimes easier to state. It says that if the practical reference is uniformly within $\bar{\delta}$ of the exact reference everywhere, then the final exact-regret certificate loses at most $\varepsilon + \bar{\delta}$ compared to the ideal exact worst case.

A.9 What the Guarantees Do and Do Not Say

For completeness, we summarize the scope of the above results. **What the guarantees establish.** The proofs show that, under A1–A4, the returned scenario \tilde{e} certifies a large avoidable performance gap for the pretrained controller. More precisely, the exact regret value $R(\tilde{e})$ is guaranteed to be close to the largest exact regret achievable over the feasible search space. The approximation error is explicit and decomposes additively into outer-search error and inner-reference error.

What the guarantees do not establish. The guarantees do not claim that the returned scenario is unique. They also do not

claim that a particular nonconvex RL algorithm always meets A3. Instead, the theorem is conditional: *if* the outer solver is ε -optimal for the implemented approximate objective, and *if* the practical reference is within $\delta(e)$ of the exact reference, *then* the returned scenario is a near-tight certificate for the exact worst-case regret over the feasible family.

Why the conditional form is still useful. This conditional structure matches how systems are analyzed in practice. The theoretical formulation makes precise what quantity the method is targeting. The theorem then shows exactly how optimization and approximation errors degrade the final certificate. As the outer solver improves or the practical reference becomes stronger, the guarantee strengthens immediately and transparently.

B Use Cases

This appendix summarizes the three controller settings used throughout the paper.

Adaptive bitrate streaming. An adaptive bitrate controller chooses the bitrate of the next video chunk at each chunk boundary. Its decision is based on the recent throughput history, the current playback buffer, and information about upcoming chunks. Pensieve [26] is an RL-based instance of this setting. Its objective is to keep video quality high over the full session while limiting rebuffering and avoiding abrupt bitrate swings.

Congestion control. A congestion-control sender adjusts its transmission behavior from transport feedback observed on the path, such as delay, loss, acknowledgments, and delivery rate. An RL-based policy in this setting makes a new control decision at the beginning of each interval, whose duration is tied to the path dynamics. Sage [40] is the RL-based congestion-control system studied in this paper. Its reward combines throughput, latency, and packet loss, so the controller must push traffic efficiently without driving the path into persistent queuing or loss.

Load balancing. A load balancer in a replicated distributed store assigns each arriving request to one of several candidate servers. The balancer observes request-arrival patterns, recent request sizes, and its own estimate of outstanding work already sent to each server, but it does not directly observe each server’s instantaneous internal utilization. Park [27] is the RL-based policy we study in this setting. Its goal is to route requests so that system-wide service remains efficient despite skewed arrivals, heterogeneous service speeds, and incomplete visibility into the servers’ real-time state.

C Additional Implementation Details

C.1 Bilevel Search Implementations for Pensieve and Park

Adaptive bitrate streaming. Pensieve [26] instantiates the same formulation in a chunk-level ABR simulator. Here the unique design choice is not process synchronization, but an exact short-horizon reference oracle. At each chunk, u_t is the bandwidth used to download the next video chunk, and the simulator updates download time, buffer occupancy, rebuffering, sleep behavior, and QoE through Pensieve’s standard dynamics. The solver controls only this bandwidth process. Pensieve still chooses bitrates through its normal observation interface, so the generated trace affects the controller only through measured throughput, delay, buffer state, next chunk sizes, and remaining chunks.

REGUARD applies Eqn. (10) exactly over a rolling window of K chunks. For the realized bandwidths and the buffer state at the beginning of the window, REGUARD enumerates candidate bitrate sequences in Π and selects the sequence with the largest QoE reward from Table 1. With six bitrate levels, this local search has size 6^K , which is tractable for small K and avoids the infeasible full-session search over 6^{48} sequences. This rolling oracle gives the outer solver dense feedback about avoidable ABR mistakes, such as over-aggressive bitrate increases before a bandwidth drop or conservative decisions that miss short high-bandwidth opportunities.

Load balancing. Park [27] instantiates the formulation as closed-loop workload generation for load balancing. At each load-balancing decision, u_t specifies the next inter-arrival time and job size, and the simulator advances queues, running jobs, completion events, and active-job-time reward after Park dispatches the current job. The action variables are typically log-scaled because both arrival gaps and job sizes span orders of magnitude. This lets the solver express bursts, idle periods, small jobs, and large jobs without leaving the controlled workload family.

Park uses a set of traditional heuristics as its portfolio for Π : Least Completion Time, Join Shortest Queue, and Choose Fastest Server. REGUARD applies Eqn. (10) by rolling each heuristic forward from the same simulator snapshot over the same generated workload window and selecting the best active-job-time reward. The distinctive implementation issue is snapshot-consistent counterfactual evaluation. The arrival stream, server speeds, current incoming job, and initial simulator state must be identical across policies, while each policy must have independent queues, running jobs, event timelines, and reward accounting. Deep snapshots enforce this separation, so a positive gap means that Park is dominated by simple decision rules under the same workload rather than by an artifact of shared mutable state. The resulting scenarios emphasize decision-sensitive workload patterns, such as bursts that make queue-length estimates misleading or size mixtures

where a poor server choice creates head-of-line blocking.

C.2 Protection Implementations for Pensieve and Park

Adaptive bitrate streaming. For Pensieve [26], the protected controller consumes the 6×8 ABR state summarized in Table 3. This state contains histories of bitrate, buffer, throughput, delay, and chunks remaining, plus the next-chunk size vector for the available bitrate choices. Pensieve’s action space is discrete because each action selects one bitrate level. For this reason, REGUARD implements the generic adjustment as a learned safe action interval rather than as a continuous nudge. The rules infer lower and upper bitrate bounds $L(s_t)$ and $U(s_t)$, which define $\mathcal{A}_{\text{safe}}(s_t) := \{a \in \mathcal{A} : L(s_t) \leq \text{bitrate}(a) \leq U(s_t)\}$. If Pensieve’s chosen bitrate lies inside this set, REGUARD abstains. If Pensieve chooses a bitrate above $U(s_t)$, REGUARD backs off by selecting Pensieve’s highest-scored action inside $\mathcal{A}_{\text{safe}}(s_t)$. If Pensieve chooses a bitrate below $L(s_t)$, REGUARD pushes harder by selecting Pensieve’s highest-scored action inside the safe set. Thus, REGUARD masks unsafe bitrate choices while preserving Pensieve’s ranking among the allowed actions:

$$a_t^{\text{prot}} := \begin{cases} a_t, & a_t \in \mathcal{A}_{\text{safe}}(s_t), \\ \operatorname{argmax}_{a \in \mathcal{A}_{\text{safe}}(s_t)} p_t(a), & a_t \notin \mathcal{A}_{\text{safe}}(s_t), \end{cases}$$

where $p_t(a)$ is Pensieve’s action score or probability for bitrate action a .

Load balancing. For Park [27], the protected controller observes the load-proxy state summarized in Table 4 before dispatching an incoming job. The original policy input contains only per-server load proxies and the incoming job size. For rule learning, REGUARD derives summaries such as total load, mean load, load imbalance, server load ranks, and each server’s load after hypothetically adding the incoming job from the same observation. These derived features are not additional inputs to the original Park policy. Park’s action space is discrete because each action selects a server. REGUARD implements protection as a risk rule followed by a per-server allow mask. The learned rules define a safe action set $\mathcal{A}_{\text{safe}}(s_t) := \{i \in \mathcal{A} : \text{AllowServer}_i(s_t) = 1\}$. If no risk rule matches, all servers are treated as safe and REGUARD abstains. If Park’s selected server is in $\mathcal{A}_{\text{safe}}(s_t)$, the action is left unchanged. If the state is risky and Park’s selected server is outside the safe set, REGUARD selects the highest-scored Park action among the allowed servers:

$$a_t^{\text{prot}} := \begin{cases} a_t, & \text{Risky}(s_t) = 0 \\ \operatorname{argmax}_{i \in \mathcal{A}_{\text{safe}}(s_t)} p_t(i), & \text{or } a_t \in \mathcal{A}_{\text{safe}}(s_t), \\ & \text{Risky}(s_t) = 1 \\ & \text{and } a_t \notin \mathcal{A}_{\text{safe}}(s_t). \end{cases}$$

This mapping realizes BACK_OFF and PUSH_HARDER as application-specific server redirections rather than scalar action changes. For example, a rule may prevent Park from sending another small job to an overloaded slow server and redirect the decision to the best Park-scored server among those

supported by reference heuristics. REGUARD therefore constrains only risky dispatches while preserving Park’s preferences whenever they are compatible with the learned safe set.

D Controller State Features

This appendix summarizes the controller state features referenced by the protection implementation in Section 5.

E Additional Evaluation Statistics

This appendix records the detailed numeric values, secondary comparisons, and per-method breakdowns omitted from Section 6.

E.1 Detailed Results for E1

For Pensieve, REGUARD reaches mean log performance gap 4.84, versus 4.00 for Random, 1.29 for Indago, 1.21 for Genet, 1.19 for Gilad et al., and 1.09 on Normal. On the original scale, that is $6.95\times$ the Random gap and $5641\times$ the Normal gap, with maximum 5.14. For Sage, REGUARD reaches mean relative performance gap 63.84%, versus 40.66% for Random and 1.66% on Normal, with maximum 70.21%. For Park, REGUARD reaches mean relative performance gap 43.49%, versus 7.05% for Gilad et al. and 0.29% on Normal, with maximum 203.80%.

E.2 Detailed Results for E2 and E3

On the most challenging family for each application, REGUARD removes 85.04% of the normalized performance gap for Pensieve, 80.01% for Sage, and 79.12% for Park. For transfer, the same REGUARD configuration removes 75.15% on Random and 22.09% on Indago for Pensieve, 90.08% on Gilad et al. and 98.54% on Genet for Sage, and 68.20% on Gilad et al. and 36.01% on Indago for Park. On Normal, REGUARD still reduces the normalized performance gap by 18.72% for Pensieve, 33.38% for Sage, and 36.11% for Park. By contrast, retraining baselines increase the Normal gap by up to 24.25% for Pensieve, approximately $15\times$ for Sage, and 81.11% for Park. Fig. 6 provides the complementary ablation result. Across all three applications, the higher-gap counterfactual sources in that figure consistently yield higher average protection percentages, while weaker sources yield much weaker protection.

E.3 Detailed Results for E4 and E5

For Pensieve, mean decision-time ratio is 3.67% without protection and 11.22%, 32.36%, 17.88%, and 27.98% across ReGuard iterations 0–3. For Sage, the corresponding values are 41.37% without protection and 50.11%, 47.55%, 49.11%, and 48.28% across ReGuard iterations 0–3. For Park, the

corresponding values are 2.00% without protection and 2.41%, 4.56%, 6.52%, and 1.85% across ReGuard iterations 0–3. Across all three systems, every iteration remains below 100% of the available decision budget, and later iterations do not induce systematic runtime growth despite stronger protection. For Pensieve, the mean gap drops from 4.84 at iteration 0 to 1.93, 1.22, and 1.24 across iterations 1–3, closing 96.6% of the distance to Normal by iteration 2. For Sage, the mean gap drops from 63.84% to 15.28%, 3.82%, and 3.78%, a 94.08% reduction by iteration 3 and a 96.59% reduction in distance to Normal. For Park, the mean gap drops from 33.45% to 6.16%, 3.21%, and 2.43%, a 92.73% reduction by iteration 3 and a 93.53% reduction in distance to Normal.

F Additional Failure Analyses

Due to space constraints, the main paper keeps only the Park case study in Section 7. This appendix contains detailed supporting statistics for Park and the corresponding Pensieve and Sage analyses. Below, pk denotes the corresponding k th-percentile predicate computed from normal scenarios (§5).

F.1 Park: Detailed supporting statistics

This subsection records the detailed quantitative support omitted from Section 7. In Fig. 9, the 107-decision slice consists of 69.2% jobs of at most 0.2KB, including 45.8% in the 0.126–0.525KB range that the fast servers can easily absorb, while only 10.3% are 5.0KB jobs appearing in four short bursts. For jobs of at most 0.2KB, Park chooses a slow server 97.3% of the time and server 1 alone 86.5% of the time, whereas Least Completion Time routes 81.1% of those same jobs to the fast tier and 71.6% to server 9. The large-gap region covers 76.6% of the slice. During that region, at least one fast server is idle on every step and server 9 is idle on every step, yet Park still sends traffic to a slow server on 86.0% of steps and to server 1 on 76.6% of steps. Inside that same region, Park never chooses a fast server, achieves mean chosen service rate 0.25 versus 0.97 for Least Completion Time, selects a zero-load server 0.0% of the time, and incurs mean regret 6.24.

Fig. 10 shows the same mechanism at workload scale. Under the challenging scenarios discovered by REGUARD, Park places 76.3% of all decisions on server 1, up from 5.6% on normal workloads. Its slow-server share rises from 29.1% to 81.5%, while its fast-server share falls from 53.1% to 14.1%. Least Completion Time under the same scenarios still routes 80.6% of decisions to the fast tier and 56.8% to server 9. At least one fast server remains idle on 98.0% of steps, yet Park still chooses a slow server with a fast server idle on 81.3% of steps and chooses server 1 while server 9 is idle on 60.8% of steps.

The recovery rules also show that reopening the fast tier is

Obs. index	Raw col.	Feature summary
0	2	Current smoothed RTT, normalized as RTT divided by 100 ms.
1	3	Current RTT variation in ms.
2	7	Current delivery rate, normalized by bandwidth factor.
3	9	TCP congestion-avoidance state.
4–12	10–18	RTT rolling summaries over short, medium, and long windows: average, minimum, and maximum.
13–21	19–27	Delivery-rate rolling summaries over short, medium, and long windows: average, minimum, and maximum.
22–30	28–36	Min-RTT-ratio and RTT-rate rolling summaries over short, medium, and long windows: average, minimum, and maximum.
31–39	37–45	RTT-variation rolling summaries over short, medium, and long windows: average, minimum, and maximum.
40–48	46–54	Inflight and unacked-data rolling summaries over short, medium, and long windows: average, minimum, and maximum.
49–57	55–63	Lost-packet rolling summaries over short, medium, and long windows: average, minimum, and maximum.
58	65	Time delta since the previous observation.
59	66	Current min-RTT-ratio or RTT-rate signal.
60	67	Current loss signal, normalized by bandwidth factor.
61	68	ACKed rate.
62	69	Delivery-rate growth ratio relative to the previous window.
63	70	Max-delivery-rate or cwnd-style utilization ratio.
64	71	Current windowed delivery rate, normalized by bandwidth factor.
65	72	cwnd-unacked ratio.
66	73	Max delivery-rate growth ratio relative to the previous max.
67	74	Max recent windowed delivery rate, normalized by bandwidth factor.
68	76	Previous Sage action or action feature.

Table 2: Sage policy observation features used by REGUARD. Sage consumes 69 selected entries from a larger 77-field shared-memory telemetry message. The raw fields excluded from the policy observation are timestamp, path or bandwidth scalar, RTO, ATO, pacing rate, slow-start threshold, one net-goodput field, and reward. In particular, reward is present in the raw shared-memory message but excluded from the policy observation.

State row	Feature	Meaning	Normalization
0	Last selected bitrate	Bitrate selected for the most recently downloaded chunk.	$s_0 = b_t / \max_b$
1	Buffer occupancy	Current playback buffer size.	$s_1 = B_t / 10$
2	Measured throughput	Throughput observed from the last chunk download.	$s_2 = S_t / (d_t \cdot 1000)$
3	Download delay	Time taken to download the last chunk.	$s_3 = d_t / (1000 \cdot 10)$
4	Next chunk sizes	Byte sizes of the next video chunk at each available bitrate level.	$s_4[a] = S_{t+1}(a) / 1000^2$
5	Chunks remaining	Number of chunks remaining in the video.	$s_5 = \min(C_t, 48) / 48$

Table 3: Pensieve policy observation features used by REGUARD. Pensieve represents state as a 6×8 matrix: six feature channels over the most recent eight decision steps. The bitrate levels are 300, 750, 1200, 1850, 2850, and 4300 Kbps. Here B_t is buffer seconds, S_t is downloaded chunk size in bytes, d_t is download delay in ms, $S_{t+1}(a)$ is the next-chunk size at bitrate action a , and C_t is chunks remaining. The observation contains history for bitrate, buffer, throughput, delay, and chunks remaining, plus the current next-chunk size vector for the six bitrate choices.

not limited to a single server. One additional correction rule is

$$(\text{Load}_9 \leq p25) \wedge (\text{PostLoad}_2 \leq p25) \\ \wedge (\text{MeanLoad} \leq p25) \wedge (\text{StdLoad} \leq p25) \implies \text{Allow}_7 = \text{true}.$$

This rule says that one fast server already looks light, another candidate would still remain light after taking the job, and the overall cluster is neither heavily loaded nor badly imbalanced. It shows that REGUARD is reopening the fast tier rather than hard-coding a single dispatch.

F.2 Pensieve: Sparse-bandwidth regimes trigger bitrate overshoot

Pensieve’s failure is sustained over-aggression under long sparse-bandwidth regimes rather than a few isolated bitrate

mistakes. Fig. 11 shows scarce-bandwidth spans that cover 35.5% of a trace duration but contain 78.6% of all 6,248 bitrate decisions. This concentration matters because the failure occupies a dominant operating regime instead of a few outliers. Within that regime, Pensieve selects 465.6Kbps on average, whereas the oracle selects 304.2Kbps, so Pensieve is 53.1% higher. When Pensieve overshoots, the overshoot is substantial rather than marginal. On the 21.3% of decisions where Pensieve exceeds the oracle, it averages 1073.0Kbps versus the oracle’s 308.8Kbps, with a mean gap of 764.2Kbps and a median gap of 900Kbps. Even inside scarce-bandwidth periods alone, Pensieve still averages 422.6Kbps versus 300.9Kbps for the oracle and chooses at least 1200Kbps on 10.4% of steps, while the oracle does so on only 0.04%. The largest mismatch reaches 4300Kbps for Pensieve versus 300Kbps

Component	Feature form	Meaning
Policy observation	$s_t = [L_0, L_1, \dots, L_{N-1}, J_t]$	Park observes one load proxy per server and the incoming job size, with $N = 10$ in the default setup.
Server load proxies	$\text{LoadProxy}_0, \dots, \text{LoadProxy}_9$	Each L_i estimates the observable load on server i .
Load-proxy value	$L_i = \sum_{j \in Q_i} \text{size}(j) + \mathbf{1}[\text{server } i \text{ is running}] \cdot \max(0, \text{finish_time}_i - t)$	The proxy combines queued job sizes with the remaining time of the currently running job.
Incoming job Clipping	IncomingSize, with $J_t = \text{size}(\text{incoming job})$ obs_high, typically 500000.0	The size of the job currently waiting to be assigned. All observation values are clipped by the observation upper bound.
Not in policy input	Future arrivals, oracle gap, reference actions, explicit queue lengths, server identities beyond vector position, and rule labels	These quantities are not part of the original Park policy observation.
Rule-derived features	Total load, mean load, minimum and maximum load, load ranks, and load-plus-incoming per server	These interpretable features are derived from the same observation for rule learning and are not extra inputs to the original Park policy.

Table 4: Park policy observation and rule-derived features. The original Park policy sees only per-server load proxies plus the incoming job size. REGUARD can derive additional interpretable predicates from the same observation without changing the policy input.

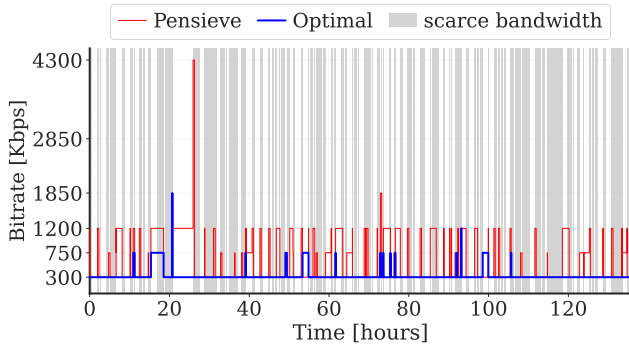


Figure 11: Sparse-bandwidth scenarios expose a systematic over-aggressive bitrate policy in Pensieve. Even when the realized link remains near the bottom of the bitrate ladder, Pensieve repeatedly selects much larger rates than the oracle, creating a large avoidable performance gap.

for the oracle under realized bandwidth of only 0.001Mbps.

The mechanism behind this failure is straightforward. Under repeated bandwidth scarcity, Pensieve keeps treating the state as compatible with aggressive upgrades even after multiple recent throughput samples collapse and chunk-download delays explode. The result is not merely a suboptimal bitrate ranking. It is a repeated decision to request chunks whose sizes the sparse link cannot support, which magnifies rebuffering and drives the large performance gap in Fig. 11.

The logic rules used by REGUARD make that mechanism explicit. One representative rule is

$$(\text{Buffer} \leq p10) \wedge (\text{Throughput}_4 \leq p10) \\ \wedge (\text{Delay}_3 > p95) \wedge (\text{NextChunk}_3 > p90) \implies \text{Bitrate} \leq 750.$$

This rule says that when the buffer is already shallow, several recent throughput samples are in the bottom decile, recent download delays are in the worst 5%, and even the next

mid-range chunk is large, Pensieve should not climb above 750Kbps. Its semantic meaning matches the failure directly: sparse bandwidth is not a one-step dip, so aggressive upgrades only deepen the mismatch between requested chunks and available capacity.

A second rule is an even harder clamp:

$$(\text{Throughput}_5 \leq p25) \wedge (\text{Throughput}_6 \leq p10) \\ \wedge (\text{Delay}_7 > p95) \wedge (\text{NextChunk}_2 > p10) \implies \text{Bitrate} \leq 300.$$

This rule says that when several consecutive throughput samples are near the floor, the most recent delay is extreme, and even moderate bitrate choices imply large chunks, the next action should be capped at the minimum bitrate. That is precisely the kind of human-readable explanation the figure calls for. The rule does not say that the network is simply bad. It says that this particular combination of sparse recent throughput, severe delay, and large pending chunks is a high-impact state in which Pensieve’s aggressive policy is likely to fail.

F.3 Sage: Sharp drops trigger overreaction and slow reopening

Sage’s failure is not generic weakness on a challenging path. As Fig. 12 shows, the dominant mechanism is an excessive reaction to bandwidth drops followed by very slow reopening. Across the plotted 50s span, Sage’s congestion window averages 0.20MB, or 139 packets, whereas BBR averages 1.56MB, or 1066 packets. The medians are 0.21MB for Sage and 1.94MB for BBR, so BBR’s window is $9.4 \times$ larger at the median and larger than Sage’s at 96.9% of sampled times. The sharpest drop, from 150Mbps to 5Mbps at 12.6s, makes Sage collapse from 261 packets to 11 packets. Sage then reaches only 50 packets after 2.45s, 100 packets after 6.3s, and half of its pre-drop window only after 9.15s. BBR also cuts back at that instant, but it rebounds to 827 packets within 50ms and

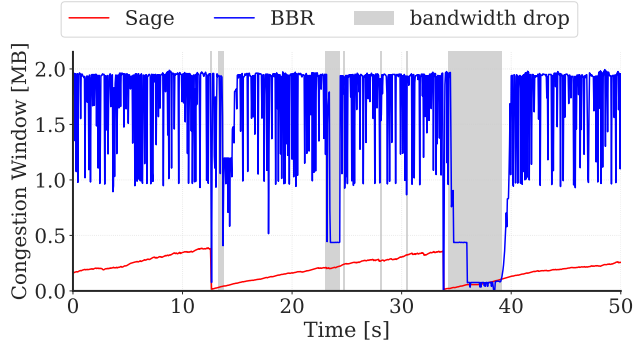


Figure 12: Sharp bandwidth drops expose Sage’s slow recovery. Sage collapses its congestion window far more than BBR and reopens it much more slowly, leaving throughput far below what the path can still support.

to 1342 packets within 100ms, regaining half of its pre-drop window in just 50ms. Over the first second after the drop, Sage averages only 19 packets while BBR averages 1097 packets, a $58\times$ gap. Even after the longest 5Mbps episode ends, BBR climbs back above 1000 packets within 0.95s of bandwidth returning to 150Mbps, whereas Sage still needs 1.85s merely to exceed 100 packets.

The key weakness is therefore not the initial backoff alone. It is that Sage continues acting as if the path remains dangerous long after the immediate drop has passed. That persistent conservatism keeps its congestion window tiny, which in turn keeps realized throughput tiny. Fig. 12 therefore exposes a specific control failure: Sage brakes hard on sudden drops and then reopens far too slowly.

The onset of that failure appears in rules such as

$$(\text{LossDelta} \geq \text{p95}) \wedge (\text{LossMin} \geq \text{p95}) \\ \wedge (\text{Rtt} \leq \text{p25}) \wedge (\text{WindowedRate} \leq \text{p10}) \implies \text{PUSH_HARDER}.$$

This rule says that loss has just surged, but RTT is still low while the windowed delivery rate has already fallen into the bottom decile. That combination is a compact signature of over-reaction. The controller has already slammed on the brakes, yet the latency signal does not justify staying that conservative.

The aftermath appears in rules such as

$$(\text{LossMax} \leq \text{p25}) \wedge (\text{LossMin} \leq \text{p25}) \\ \wedge (\text{MaxRate} \leq \text{p10}) \wedge (\text{AvgRate} \leq \text{p10}) \implies \text{PUSH_HARDER}.$$

This rule is important because it no longer describes an acute crisis. Loss is now low, RTT inflation is not severe, and yet both the recent maximum and the recent average delivery rates remain near the floor. In plain language, the network is no longer screaming “back off,” but Sage is still behaving as though it is.

A third rule isolates the slow-recovery phase even more directly:

$$(\text{CurrentRate} \leq \text{p10}) \wedge (\text{RateGrowth} \leq \text{p10}) \\ \wedge (\text{AvgRate} \leq \text{p10}) \wedge (\text{RateVsMax} \leq \text{p25}) \implies \text{PUSH_HARDER}.$$

This rule says that Sage is not only small in absolute terms. It is growing slowly and operating far below what it was recently capable of. That is exactly the behavior seen in Fig. 12, and it makes the failure interpretable in a way that the raw congestion-window trace alone cannot.

G Testbed

For fair evaluation, we conduct all experiments on a two-socket server with 40 logical Intel Xeon E5-2660v3 CPUs running at 2.60GHz and 256GiB of DRAM.