USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# Controlling Arbitrary Internet Queues with Titrate

Anchengcheng Zhou and Joshua Lau, *Princeton University;*
P. Brighten Godfrey, *University of Illinois Urbana–Champaign and Broadcom;*
Maria Apostolaki, *Princeton University*

https://www.usenix.org/conference/nsdi26/presentation/zhou-titrate

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology


# Controlling Arbitrary Internet Queues with Titrate

Anchengcheng Zhou
*Princeton University*
*ann.zhou@princeton.edu*

Joshua Lau
*Princeton University*
*motoaki@princeton.edu*

P. Brighten Godfrey
*UIUC and Broadcom*
*pbg@illinois.edu*

Maria Apostolaki
*Princeton University*
*apostolaki@princeton.edu*

## Abstract

Router buffers are critical to networks for absorbing short-lived congestion and allowing full throughput. However, excessive buffering can lead to high queuing delay, poor burst absorption, and even low throughput for queues sharing the buffer memory. Existing queue management schemes designed for Internet routers (*e.g.,* CoDel, PIE) prevent such excessive buffering only under stringent assumptions about the queue composition (flows in the queue), while more recent approaches (*e.g.,* L4S) require end-host collaboration. In this work, we revisit queue management for Internet routers from first principles and introduce Titrate, a closed-loop controller that senses queue dynamics and adjusts thresholds for any given queue to achieve high throughput, low latency and effective burst absorption. To balance convergence speed and stability, Titrate draws inspiration from TCP's control loop, combining a multiplicative-increase-additive-decrease approach with an ssthresh-like variable.

We evaluate Titrate's performance via simulation and Internet experiments. Across a wide range of realistic traffic mixes, Titrate increases minimum throughput by 39%, 14% compared to CoDel, PIE, while keeping 59% lower queuing latency compared to static-threshold baselines of on-par throughput. It also improves end-user quality of experience over static-threshold baselines. We further show that Titrate reacts swiftly to bandwidth and traffic changes and offers device-wide benefits.

## 1 Introduction

Router buffers, where data packets are temporarily stored to avoid loss during short-lived congestion, are ubiquitous in the Internet and essential for performance. First, buffers allow queues to reach full throughput. Critically, buffers are necessary not only for loss-based congestion control algorithms (CCAs) that direct the majority of Internet traffic [33] but also for rate-based and delay-based CCAs that still need a small (but non-zero) amount of buffer for sustaining high throughput. Second, buffers improve user experience for bursty applications such as web browsing and video streaming. Web services often open multiple flows or use larger initial windows to quickly grab bandwidth and shorten page load time [18, 32], hence creating bursts. Similarly, video traffic is bursty because it transmits data segment by segment.

However, excessive buffering (*i.e.,* using more buffer memory than necessary to sustain throughput and absorb application bursts) harms performance. Every unnecessary packet buffered increases latency for delay-sensitive applications, delays congestion feedback to senders, and reduces the buffer's ability to absorb bursts (which are critical to application performance). Using more buffer memory than necessary on one port can even deprive other ports in the same router of the buffer memory they need for full throughput or burst absorption, as buffer memory is shared [6, 7].

Unfortunately, ensuring that a queue on an Internet router operates at its ideal point (*i.e.,* using just sufficient buffer memory to sustain throughput and absorb application bursts) is extremely challenging. This ideal point varies wildly across different queues, because it depends heavily on the queue composition, that is, the specific set of flows the queue carries. CCAs, round-trip times (RTTs), and application behavior are only a subset of factors influencing queue dynamics and, consequently, where the ideal point is. This is further complicated by the fact that queue compositions on the Internet are both highly diverse and constantly changing.

Existing approaches perform well only for a narrow set of queue compositions, which naturally lead to more predictable queue dynamics, but they fail to generalize to the diverse and unpredictable queues seen in Internet routers. For example, buffer-sizing theory [12, 31, 39] can estimate the buffer needs of a queue when it carries only flows running a single, well-defined CCA. Buffer sharing solutions [8, 9] that calculate queue thresholds are designed for datacenters where queue compositions tend to be less diverse due to clear traffic classification or/and a limited set of transport protocols. L4S [13] sidesteps the diversity and unpredictability of queue dynamics by assuming that traffic can be split into two homogeneous queues with predictable behavior—yet

this relies on accurate end-host marking. On the open Internet, this is difficult to guarantee, and even a single mis-marked flow can undermine the intended benefits of low latency. Earlier AQMs such as Codel [26] and PIE [36, 37] address the problem head-on but were designed at a time with much less CCA diversity and even less traffic altogether. As a result, they do not generalize to the more complex and varied queue dynamics found in the modern Internet.

In this paper, we take a first-principles approach to determining the threshold (*i.e.,* the maximum allowable queue length) that enables a queue to operate at its ideal point, achieving full throughput with minimal latency while accommodating application bursts. We introduce Titrate, a closed-loop controller that continuously infers how the current threshold affects queue behavior—and, by extension, flow performance—by sensing queue dynamics, and adjusts the threshold accordingly. Designing such a controller is inherently challenging. Queue dynamics on the Internet are highly diverse, noisy, and often ambiguous, making it difficult to determine how to adjust the threshold. Adding to the complexity, the ideal threshold that the controller aims to maintain shifts constantly with the ever-changing queue composition. A sluggish control loop may fail to find appropriate thresholds in time before the queue composition changes, while an overly aggressive loop can induce instability and oscillations. To address these challenges, Titrate obtains a more comprehensive view of queue behavior by monitoring both the minimum queue length and the duration of zero-queue periods only after packet drops. Drawing inspiration from TCP, Titrate adjusts the queue threshold following a multiplicative-increase, additive-decrease strategy and also incorporates an ssthresh-like variable to balance rapid convergence with long-term stability. Finally, Titrate treats traffic bursts as outliers, allowing them to pass without perturbing the state of the control system. Importantly, Titrate is designed to operate effectively across a wide range of queue compositions, making it broadly applicable in Internet settings.

Both ns-3 simulation and Internet experiments [1] demonstrate that Titrate achieves high throughput, low latency and effective burst absorption across a wide range of queue compositions. Ns-3 simulation shows that across diverse, realistic traffic mixes, Titrate achieves 38.49% [2] and 13.62% higher minimum throughput than CoDel and PIE, respectively. Enforcing a static threshold of one bandwidth-delay-product (BDP) achieves the same throughput as Titrate, but Titrate reduces queuing latency by 59.04%. Furthermore, on devices with small buffer memory, Titrate offers throughput improvement and a staggering 90.36% reduction in burst completion time, compared to DT, the default buffer management mechanism on routers today [7, 8, 17, 19]. Titrate also swiftly and efficiently adapts to changing network and traffic conditions. Additionally, our testbed evaluation on Internet video traffic and file downloads has shown that Titrate offers better end-user quality of experience across a wide range of Titrate parameterizations than even well-configured static thresholds.

**Why bother and why buffer?** We acknowledge two points of skepticism that may concern the reader, and address them directly. The first is that congestion is no longer a pressing issue in today's Internet, and hence buffers rarely fill to the point that affects latency. However, the exponential growth of data traffic continues to push the limits of network infrastructure, making congestion not just possible but inevitable. Scaling up bandwidth is theoretically feasible, but it incurs significant cost, borne by ISPs that do not directly profit from application-level performance gains. Consequently, continual upgrades are economically unsustainable, as illustrated by peering disputes [3]. Overprovisioning buffers may temporarily mask the problem, but introduces substantial latency, energy consumption, and operational overhead. Worse yet, deep buffers are increasingly impractical due to hardware constraints and limits imposed by Moore's Law [30].

The second point is that people may think that the throughput-latency tradeoff is best left to end-host CCAs. Yet in shared networks, buffer contention across flows leads to a tragedy of the commons. This is not hypothetical: TCP Cubic, despite its age, remains dominant [33], prompting newer CCAs to become more aggressive in order to remain competitive. Meanwhile, end hosts lack sufficient visibility into in-network conditions, and the diversity of queue dynamics makes effective decision-making even harder for them. Introducing adaptive, in-network mechanisms offers a complementary path forward—making performance more predictable and robust under realistic Internet conditions.

## 2 Motivation & Related Work

To motivate the need for a scheme that controls queue lengths at Internet routers, we begin with describing a concrete pain point faced by network operators. We then distill the key properties that a solution must satisfy and explain why existing approaches fall short.

### 2.1 Motivating Example

Consider an operator of an edge network who observes their border-gateway router being increasingly more congested in multiple ports at a time, a direct consequence of the ever-increasing volume of Internet traffic. What is more alarming is that during congestion, the quality of experience plummets: there is excessive queuing delay, bursts are severely dropped, and even throughput on some ports can suffer due to buffer pressure [10]. While upgrading bandwidth would solve the problem, it is not a sustainable solution, and while upgrading to deeper buffers could reduce drops, it will further degrade

[1] Our artifacts have been made publicly available at https://github.com/AnnZhouCcc/Titrate.

[2] The next three numbers (this included) are a combined result from Figure 8 and Figure 9 for comprehensiveness.

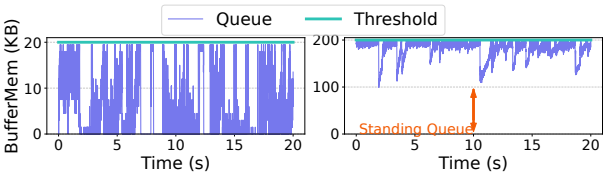



Figure 1: Under-buffering leads to throughput loss (Left) while over-buffering leads to queuing delay (Right).

latency. Asking senders to be less aggressive or to mark their traffic to facilitate prioritization is unrealistic, as an operator neither controls nor trusts them. The only practical option for the operator is to constrain each queue's growth by setting up a threshold. Doing so should eliminate long queuing delays, alleviate buffer pressure and leave the buffer ready to absorb bursts.

While intuitive, naively setting up a threshold can backfire on the operator, as both over- and under-buffering are detrimental to performance. As an illustration, consider the simple example in Figure 1 showing the same queue with two different thresholds. When the threshold is 20KB, throughput is only 84%, as there is not sufficient headroom to keep the link busy when senders ramp up their windows (Figure 1 (Left)). When the threshold is 200KB, a standing queue of 8ms (marked in orange) is formed, causing unnecessary delay on the order of the flows' RTT for packets in this queue (Figure 1 (Right)).

Clearly, there exists a threshold that avoids both over- and under-provisioning for the queue in Figure 1. However, that threshold differs depending on the queue composition. Queues do not occupy a fixed amount of buffer over time; instead, their length fluctuates with the flows' aggregate congestion window [12,31]. As buffer sizing theory [12,24,31] points out, the ideal queue threshold *just* accommodates queue oscillations, ensuring high throughput while minimizing queuing latency. In Figure 2, we demonstrate the queue oscillations over time for four queues of different compositions (in terms of CCAs, RTTs and number of flows) under a fixed threshold (which is sufficiently large for full throughput). We observe that queue oscillations vary drastically in amplitude and frequency even across these four queue compositions, which represent only a small subset of queue diversity seen on the Internet. Consequently, the ideal queue threshold, *i.e.,* the minimum buffer memory required to accommodate these queue oscillations (marked in orange) also varies, making it challenging to decide on a single threshold for arbitrary queue composition.

## 2.2 Requirements

The previous example reveals the need for a scheme that controls queue lengths and satisfies three key requirements: *(i)* performance; *(ii)* generality; and *(iii)* practicality. **Performance** refers to the need for high throughput, low latency and effective burst absorption for supporting the diverse Internet applications. Over- and under-buffering would fail this requirement. **Generality** refers to the need for supporting queues of arbitrary compositions without prior knowledge. Finally, **practicality** refers to the need for a scheme that requires neither hardware upgrades nor end-host cooperation (*e.g.,* marking).

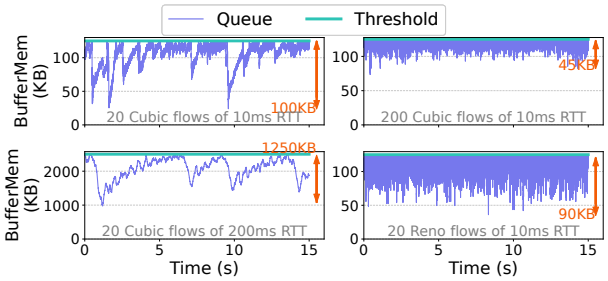



Figure 2: Queue dynamics vary drastically, in both oscillation amplitude and frequency, depending on CCAs, RTTs and number of flows. Consequently, the minimum buffer required by each queue for full throughput and low latency (marked by the orange arrows) also varies drastically across the four queue compositions.

## 2.3 Limitations of Existing Work

Multiple lines of prior work have looked into managing queues. We will discuss how they fail to adequately address the problem and meet our requirements above.

**Analytical solutions require perfect knowledge of queue compositions and do not generalize to arbitrary queues.** Buffer sizing theory [12, 24, 31] analytically derives the ideal thresholds for queues to achieve high throughput with minimum latency. However, these derivations require precise knowledge of the queue composition and, in practice, only work for queues that carry flows using a single, well-documented CCA. Queues at Internet routers, however, carry flows using highly diverse and often undocumented CCAs, making it impractical to model them precisely and derive thresholds analytically. Even if some approximate modeling was adequate to derive meaningful thresholds, re-calculating thresholds on-demand to follow the dynamically changing composition of queues is not realistic.

**Grouping traffic into homogeneous queues is useful but of questionable practicality, hence orthogonal to this work.** Recognizing the challenges of controlling queues of arbitrary composition, many works sidestep the problem by splitting traffic into queues of more constrained compositions and then controlling these homogeneous queues. This approach is particularly effective in datacenters, where traffic is more controlled and splitting traffic into queues of pre-constrained compositions is possible. Recent buffer sharing algorithms designed for datacenter settings [8, 9] improve upon DT [17] (the default buffer management scheme) by tailoring controls on each queue of pre-constrained compositions to that particular composition. Indeed, ABM [8] assumes queues of pre-determined CCAs that are of either high priority, meaning they only see bursts, or low priority, meaning they are composed of long flows that need throughout.[3] Similarly,

[3]The single queue setting in ABM still assumes a flow classifier.

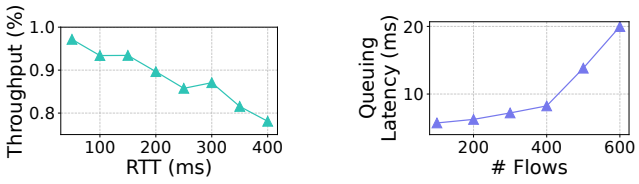


Figure 3: Because of its simplistic assumptions about queue dynamics, CoDel suffers from severe throughput loss when the average flow RTT in the queue increases (Left) and from queuing latency inflation when the number of flows in the queue increases (Right).

Reverie [9] assumes queues of lossy and lossless traffic. While similar approaches exist for Internet queues, their practicality is questionable. For instance, L4S [13] assumes that hosts will correctly mark their L4S-enabled flows, and the L4S-enabled flows will be isolated in a separate queue to avoid latency inflation in the dual-queue AQM system. However, just a single mis-marked non-L4S-enabled flow can substantially degrade the performance of the L4S queue (see Figure 11).

**Traditional AQMs rely on rigid heuristics for identifying standing queues, which do not generalize.** Active queue management schemes (AQMs) can deliver high throughput and low latency, but rely on simple heuristics to detect over-buffered queues, which only work for specific compositions that create specific queue dynamics. As a result, such solutions do not generalize to arbitrary Internet queues, as we show in §5.2. Next, we discuss a few representative AQMs in more detail and explain how their assumptions on queue dynamics fall short.

RED [21], the well-known AQM that drops packets when the average queue size exceeds a preset *target*, implicitly assumes that any queue with a larger average queue size is over-buffered. It requires very careful parameter tuning for different network conditions and thus poses significant challenges in deployment [26, 36, 37]. Moreover, RED controls queue length rather than queuing latency directly, and can be problematic on ports with multiple queues whose bandwidths can vary. PIE [36] is another AQM that detects the onset of congestion based on whether the queuing latency is above a preset *target* and whether the latency is increasing or decreasing, and drops packets accordingly. Similarly, PIE assumes that a good queue should not have queuing latency beyond *target*, but as previously discussed, queues with different traffic compositions exhibit remarkably different queue dynamics, making a one-size-fits-all *target* value overly simplistic and potentially ineffective. PI2 [20] improves PIE's control law and stability, but still relies on a single *target* delay.

CoDel [26] improves upon RED and drops packets when the minimum packet sojourn time exceeds *target* for *interval* time. However, this assumption of good queue dynamics fails to generalize too. As Figure 3 (Left) shows, throughput decreases as RTT increases. This happens because large-RTT flows induce larger oscillations, so the minimum packet sojourn time stays above *target* for longer even when the queue is healthy. However, CoDel's rigid assumption on good queue dynamics deems this problematic and over-controls the queue, leading to throughput loss. Furthermore, CoDel's control law is ineffective for queues with many flows. In Figure 3 (Right), queuing latency increases as the number of flows increases. This happens because a queue with many flows needs more packet drops than a queue with a single flow to inform a significant proportion of the senders to slow down. CoDel's baked-in control law assumes a particular response in queue dynamics to its actions, so under many senders, it misestimates the amount of packet drops needed for the senders to back off and fails to control queue growth. COBALT [35] improves CoDel's control law and reduces the latency inflation for queues with more flows. But it still suffers from the same issue of throughput loss with longer-RTT flows (see Figure 18 in Appendix 8.1) and sometimes even with short-RTT flows (see Figure 8).

## 3 Titrate Overview

Having explained why controlling queues of arbitrary composition is challenging and unresolved by existing work, this section provides an overview of Titrate. We start by discussing the insights that drive Titrate's design (§3.1) before providing an end-to-end view (§3.2).

### 3.1 Challenges & Insights

From first principles, our goal is to design a control loop that dynamically adapts the queue threshold to achieve full throughput with the smallest possible buffer memory while absorbing bursts. To do so, we need to answer three fundamental questions: *(i)* What should the control system sense, and how should it interpret those signals to infer the effect of the current threshold on queue performance? *(ii)* How should the threshold be adjusted to strike a balance between responsiveness and stability? *(iii)* How can the system distinguish and ignore noise, *i.e.,* signals that are unrelated to the control variable? Next, we highlight our insights for answering these questions.

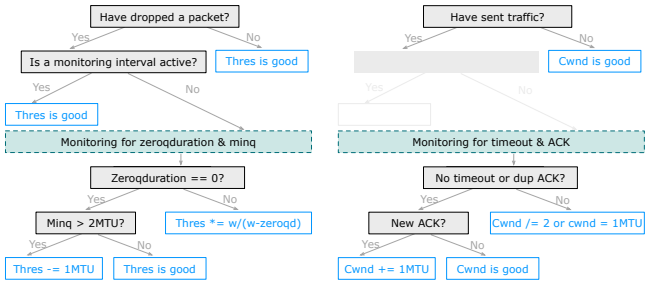
**Jointly monitoring zero-queue duration and minimum queue length offers a comprehensive view of throughput and latency for any queue. (§4.1)** It is crucial to understand how a threshold affects a given queue before deciding how to adjust it. While switches expose many runtime statistics *e.g.,* enqueue/dequeue rates, queue occupancy, not all of these signals are informative or reliable, and none alone provides unambiguous feedback for the performance of arbitrary queues. For example, minimum queue length does not capture the severity of throughput loss, while total packet count or average queue length cannot identify a standing queue causing unnecessary delay. Instead, Titrate monitors both the duration of queue length reaching zero for a fine-grained estimation of throughput, and whether the minimum queue length exceeds a conservative safe threshold (*e.g.,* 2MTU) for detecting standing queues.

**Collecting feedback just after a packet drop filters out noises and improves signal quality. (§4.1)** Queue dynamics

are affected by factors beyond the threshold adaptations, which are effectively noise from the perspective of the control loop and need to be distinguished from genuine sender-side feedback to threshold adaptations. Such factors include bandwidth fluctuations on downstream links and application-level throttling. For example, extended zero-queue duration may occur either because the threshold is too low and hurts throughput, or because the senders have little data to transmit. Hence, reacting to the zero-queue by increasing the threshold may cause unnecessary oscillations. To increase the chances of collecting genuine feedback, Titrate only monitors signals after observing a packet drop. Critically, a drop affects how senders view network conditions and thereafter affects the performance of the queue. By collecting feedback just after drops, Titrate can capture the reaction of the senders to the current threshold. This design choice has an interesting and useful implication for self-controlled queues such as those under latency-sensitive CCAs. Because Titrate is triggered by drops, it does not interfere with flows that eagerly identify the maximum rate they can send at without causing unnecessarily long queues.

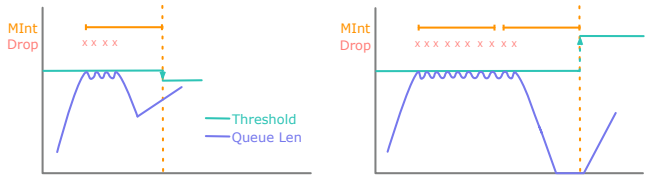
**Multiplicative-increase-additive-decrease and keeping an ssthresh-like variable ensure fast yet stable convergence to the right threshold. (§4.2)** Having gained an accurate understanding of how a threshold performs, the next challenge for our control loop lies in dynamically adjusting the threshold such that it can quickly respond to performance issues without risking stability. Blindly making large adjustments risks overshooting which induces oscillations and might prevent convergence, while always making conservative, small adjustments risks slowing down convergence or/and dooming the system to chronic under-performance as network conditions evolve. We observe that not all threshold adjustments carry the same risk nor urgency. Hence, Titrate adjusts more aggressively when increasing the threshold (lower risk) to avoid hurting throughput (higher urgency) and more conservatively when decreasing it (higher risk) to avoid unnecessary delay (lower urgency). Specifically, if the queue remains at zero length for an extended period after experiencing drops, Titrate can safely infer that the threshold is too low and increase it multiplicatively to rapidly restore performance. In contrast, if the minimum queue length is high, Titrate cannot infer conclusively whether the threshold is too high, as the queue length might drop further given more time. To avoid overreacting, Titrate applies an additive decrease, cautiously lowering the threshold while maintaining stability. This is a parallel design principle to TCP, where a packet loss is a less ambiguous signal than the reception of a single acknowledgment and avoiding congestion collapse is a more important target than increasing the sending rate.

Meanwhile, while stability is important, reducing the threshold too slowly can lead to unnecessary queuing and increased application latency. To mitigate this, Titrate maintains a state variable analogous to TCP's slow-start threshold (ssthresh). Instead of always applying an overly cautious additive decrease, Titrate rapidly decreases threshold




(a) Titrate framework (b) (Simplified) TCP framework

Figure 4: Titrate faces challenges similar to TCP in interpreting and responding in a noisy environment, so it adopts a similar framework. We omit MI-thresh and ssthresh for clarity.




(a) Threshold decrease (b) Threshold increase

Figure 5: Illustrations of how the threshold changes based on the zeroqduration and minq signals on a fixed-length monitoring interval.

until it reaches "ssthresh" and then switches to additive decrease, allowing it to balance responsiveness with stability.

**Ignoring temporary spikes in queue lengths helps absorb bursts without affecting the control system. (§4.3)** In addition to accommodating queue oscillations to deliver full throughput, buffer also plays a vital role in absorbing application spikes (bursts) to deliver low application latency. However, bursts are random, non-actionable feedback, so they should be absorbed without confusing the control system (*i.e.,* without being misinterpreted as signals that warrant threshold changes). We observe that application spikes appear as outliers in their queue length patterns, as they are typically shorter-lived than queue oscillations from long-lived flows. Hence, instead of precisely detecting them, Titrate ignores them using a lightweight outlier filter. We believe this approach is more practical for the Internet compared to isolating bursts on an independent queue after detecting through per-flow runtime statistics (*e.g.,* IB [11, 40]).

## 3.2 Lessons from TCP & Design Overview

Figure 4a illustrates how Titrate determines the queue threshold). When a packet is dropped, Titrate starts a monitoring interval if no interval is currently active. During the monitoring interval, Titrate keeps track of the duration of queue length being zero (zeroqduration) and the minimum queue length (minq) (§4.1). At the end of the monitoring interval, Titrate first checks whether zeroqduration is zero. If it is non-zero (*i.e.,* the queue drained at some point after the drop), Titrate increases the threshold multiplicatively, in proportion to a ratio

between interval length and zeroqduration. If zeroqduration is zero and minq is larger than 2MTU, Titrate decreases the threshold additively. Otherwise, Titrate leaves the threshold unchanged (§4.2). To absorb bursts, Titrate only drops a packet if the filtered average queue length (or effective queue length, EQlen) and instantaneous queue length are both above the queue threshold (§4.3).

Titrate as a control system draws inspiration from TCP, as Figure 4 illustrates. Titrate observes queue dynamics to decide the threshold for each queue, as TCP observes congestion signals to decide the congestion window for each flow. They face similar challenges in interpreting extremely noisy environment, deciding how to respond and balancing agility and stability. As a result, they react to different signals by gauging a risk/urgency (throughput loss for Titrate vs congestion collapse for TCP) to performance gains (higher sending rates for TCP vs lower average latency for Titrate). They also react more decisively to some signals (zero-queue duration for Titrate vs packet loss for TCP) than others that might need more time (minimum queue length for Titrate vs receiving an ACK for TCP).

# 4 Design

This section delves into the design details of Titrate's control system: the signals it senses (§4.1), the interpretations it derives from the signals (§4.2) and lastly, the resulting actions (§4.3). We discuss Titrate's practicality in §4.4 and its design parameters in Appendix §8.3.

## 4.1 Signals

**Monitoring multiple signals.** Switches have the unique capability of observing queue statistics during runtime. With the right signals, we can gain an accurate understanding of how throughput and queuing latency perform in response to the current threshold. In Titrate, we monitor (1) the total duration when queue length reaches zero, or *zero queue duration (zeroqduration)*, for throughput and (2) the *minimum queue length (minq)* for queuing latency. Both signals satisfy three properties that make them appropriate signals to sense: fine-grained, bandwidth-independent and queue-independent. A fine-grained signal indicates the severity of performance degradation rather than just its existence, and informs how large a threshold adjustment to make. A bandwidth-independent and queue-independent signal has universal target values that do not vary across different bandwidth links or queue compositions, allowing the scheme to work across diverse scenarios. Table 1 in Appendix 8.2 presents two sets of candidate signals for throughput and queuing latency, respectively, and demonstrates how zeroqduration and minq meet all three requirements while others do not.

Using both zeroqduration and minq creates a tension in choosing the monitoring interval. While both need to be aggregated over a reasonable interval, zeroqduration prefers a shorter interval so that once it is non-zero, we can recover from the throughput loss fast; on the other hand, minq prefers a longer interval so that we can aggregate over more data for a more reliable minq signal. In Titrate, we navigate this tension by implementing an optimization that triggers an additional monitoring interval if $zeroqduration = 0$ *i.e.,* no throughput loss has incurred when the first interval ends. At the end of the second monitoring interval, we follow normal procedure and react to either throughput loss from the second interval or a more reliable minq signal that is aggregated over two intervals.

**Drop-triggered monitoring.** The network is a fundamentally noisy environment with many concurrent, unpredictable events. Gaining an accurate understanding of a threshold's performance crucially depends on sensing genuine feedback instead of network noises *e.g.,* non-threshold-induced events. In Titrate, we start a monitoring interval upon a packet drop if no interval is currently active, to maximize our chances of observing feedback from senders on the current threshold. At most one monitoring interval can be active at any time. Packet drops are a useful trigger because most senders, upon detecting a packet drop, make some interpretations out of this drop event and adapt their sending rates accordingly. Thus, by monitoring queue dynamics immediately after a drop, Titrate can capture how senders respond to the threshold after actually learning about where the threshold is. This strategy works well for drop-reactive senders (*e.g.,* flows running loss-based CCAs like Cubic). For non-drop-reactive but latency-sensitive senders (*e.g.,* flows running delay-based CCAs like Copa) that naturally keep their buffer usage small, Titrate does not intervene by design. For non-drop-reactive and buffer-aggressive traffic (*e.g.,* UDP flows), Titrate monitors after drop events and observes that high throughput persists even under small buffers, and consequently reduces their queue thresholds.

**RTT-independent monitoring duration.** The monitoring duration is set to be a constant independent of the average flow RTTs in a queue. As Figure 5 shows, packet drops often occur in succession as sender detection and reaction take time, and some senders respond only after multiple drops. Consequently, after the first packet drop and the monitoring interval starts, queue length typically hovers near the threshold for a short period. If this process takes less than the monitoring duration, we observe the sender response within the interval and adjust the threshold according to either minq (Figure 5a) or zeroqduration. If this process takes longer than a monitoring interval, we would have started a new monitoring interval and captured the sender response in that later interval (Figure 5b). While it is still possible to miss the sender response if the hovering takes even longer, we find that in practice a constant monitoring duration performs well across diverse queue scenarios.

## 4.2 Interpreting the Signals

**Direction of threshold adjustment.** At the end of each monitoring interval, we observe *zeroqduration* and *minq*

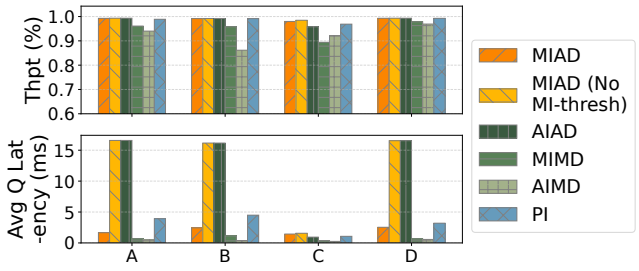



Figure 6: Titrate achieves both high throughput and low queuing delay across diverse queue compositions, outperforming other threshold adjustment strategies. Queue compositions: (A) 100 100ms(RTT)-Cubic flows, (B) 10 100ms-Cubic flows, (C) 100 10ms-Cubic flows, (D) 100 100ms-Reno flows.

and first determine whether to adjust the threshold, and if so in which direction. Titrate leaves the threshold unchanged when it infers both high throughput and low latency *i.e.,* when $zeroqduration = 0$ (the queue never drains) and $minq \leq 2MTU$ (the minimum queue is small). If $zeroqduration > 0$, the queue has already suffered throughput loss, indicating that the threshold is too low, so we increase it. Otherwise, when $zeroqduration = 0$ and $minq > 2MTU$, we infer a standing queue and decrease the threshold.

**Magnitude of threshold adjustment: multiplicative-increase-additive-decrease (MIAD).** The size of each threshold adjustment in a given direction is crucial for converging to the ideal threshold quickly. Blindly taking large adjustments risks overshooting and inducing oscillations around the target threshold, prolonging performance degradation. However, small adjustments can be too conservative to recover quickly from performance degradation. The key insight that enables Titrate to converge to the ideal threshold quickly is that different adjustments carry different risks and we are more conservative with higher-risk adjustments and more aggressive with lower-risk ones. $zeroqduration > 0$ is an unambiguous signal suggesting that the threshold is too low, and thus we increase the threshold more aggressively. On the other hand, a minq signal is less conclusive in suggesting whether the threshold is too high or not, given that the minq value could decrease further with a longer monitoring duration. Thus, we decrease the threshold by a small value. Figure 6 compares different threshold adjustment strategies, including a proportional-integral (PI) controller, in response to zeroqduration and minq signals across four different queues. AIAD is conservative with both threshold increase and decrease, incurring high queuing latency for queues A,B,D due to conservative threshold decrease and also throughput loss for queue C due to conservative threshold increase. Both MIMD and AIMD are aggressive with threshold decrease and incur low throughput. The PI controller achieves decent performance but still incurs mild throughput loss for queue C and longer queuing latency for others.

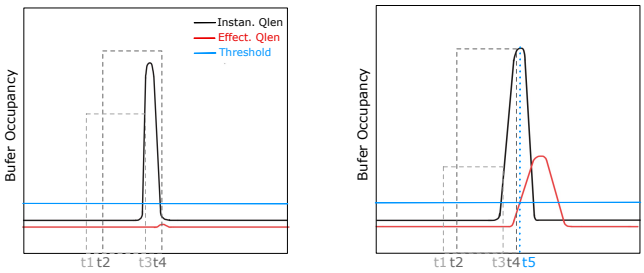
Moreover, we increase the threshold proportional to how much throughput loss has occurred, as indicated by the value of zeroqduration. Specifically, given a monitoring interval of length $w$ and a zeroqduration signal of value $x, x > 0$, we increase the threshold multiplicatively: $thres_{new} = M * thres_{old}, M = c_{inc} * \frac{w}{w-x}$, where $c_{inc}$ is a constant (Appendix §8.3). This formulation of $M$ has three useful properties. First, for $x > 0$, $M > 1$, ensuring that we will always increase the threshold when throughput loss is observed. Second, $M$ increases with $x$; when $x$ is small (little throughput loss), the threshold increase is small, whereas when $x$ approaches $w$ (severe throughput loss), $M$ becomes large and the threshold increase is also large. Third, $M$ is convex in $x$, ensuring that as throughput loss worsens and $x$ increases, $M$ grows more than linearly to quickly recover from the throughput loss. In comparison, we decrease the threshold conservatively by a small value : $thres_{new} = thres_{old} - c_{dec} * 1MTU$ where $c_{dec}$ is a constant (Appendix §8.3).

**Speeding up the additive-decrease: MI-thresh.** The current threshold can be arbitrarily far from the ideal, so always decreasing it by a fixed $c_{dec} * 1MTU$ is suboptimal, especially when we have high confidence that the current threshold is considerably larger than the ideal. Titrate borrows the idea from TCP's ssthresh that we keep track of an estimate of a safe threshold, termed *MI-thresh*, above which we believe there is a very small chance of incurring throughput loss. MI-thresh is calculated based on $thres_{prevMI}$, the threshold under which the queue experiences multiplicative increase the previous time. Specifically, $MI\text{-}thresh = c_{thresh} * thres_{prevMI}$, where $c_{thresh}$ is a constant (Appendix §8.3). When we decide to decrease the threshold at the end of a monitoring interval, we compare the current threshold $thres_{old}$ with MI-thresh. If $thres_{old} > MI\text{-}thresh$, set $thres_{new} = \frac{1}{2}(thres_{old} + MI\text{-}thresh)$. Otherwise, set $thres_{new} = thres_{old} - c_{dec} * 1MTU$. This strategy enables fast convergence when the threshold is clearly too large, while maintaining stability nearer the target. Figure 6 compares MIAD with and without MI-thresh and demonstrates that employing MI-thresh ensures low queuing latency without compromising high throughput.

## 4.3 Acting upon the Interpretations

In the previous subsections, we have determined the direction and magnitude of threshold adjustments based on observed runtime signals. Next, we discuss how to close the decision loop of whether to enqueue or drop a packet by comparing the threshold with the computed queue length.

**Computing filtered average queue length: effective queue length (EQlen).** When deciding whether to enqueue or drop a packet, it is crucial to distinguish bursts from long-lived flows. Packets in a burst should be enqueued whenever possible because bursts are transient (therefore do not sustain persistent queue growth) and are latency-sensitive. In contrast, packets from long-lived flows are enqueued or dropped according to the queue threshold. A key challenge here is to correctly identify *true* short bursts versus burst-like spikes from long-lived flows. We must retain these burst-like spikes, since




(a) An actual burst (b) A spike of long-lived flow

Figure 7: Illustrations of how EQlen differentiates between an actual burst and a burst-like spike of a long-lived flow, allowing the former to pass but controlling the latter.

they generate drop signals that drive threshold adaptations.

Titrate leverages two characteristics of bursts — (1) they cause rapid queue build-up and (2) they are short-lived — and detects bursts as outliers and therefore exempts them. Concretely, Titrate samples queue length over a window where burst samples remain a minority, discards any sample exceeding the window's simple average by more than a tolerance, and computes *EQlen* as the mean of the remaining samples. This design hinges on the insight that burst-like spikes from long-lived flows will keep growing and eventually become the majority in a window. At that point, they inflate the simple average, cross the threshold and thereafter get dropped. Fig. 7a shows a burst that will escape thresholding, while Fig. 7b shows a burst-like spike of a long-lived flow that will eventually experience packet drops. The window over which we aggregate queue length data points is of length $t_3 - t_1 = t_4 - t_2$. At time $t_3$, both the burst and the burst-like spike are considered outliers, and thus EQlen stays low in both cases. At time $t_4$, in Fig. 7a, the window is large enough such that most of the burst queue lengths are categorized as outliers and excluded from the EQlen computation. EQlen sees some minor increase, which is below the threshold and the entire burst is effectively ignored by the threshold and thus be enqueued in the buffer. In Fig. 7b, however, since the spike comes from a long-lived flow, the queue length would keep increasing until most of the spike queue lengths become the majority in the window and thus be included in EQlen. At this point, EQlen starts to increase, which eventually crosses the threshold at time $t_5$. Packets from the flow start to get dropped, and the queue length starts to decrease, thereby preventing a spike from a long-lived flow from growing without control.

**Avoiding over-penalizing queue spikes.** Titrate decides to drop a packet only when both the instantaneous queue length and EQlen exceed the queue threshold. We intentionally avoid dropping based on EQlen alone: an increase in instantaneous queue length will impact EQlen a few time steps later, therefore a past queue build-up can keep EQlen elevated even after the queue itself has subsided. In such cases, when EQlen is above the threshold but the instantaneous queue length is below, we do not drop packets, avoiding penalizing the current traffic for past queue conditions.

### 4.4 Practicality

While the switch MMU is not open to programmers even on programmable switches [38], Titrate can be implemented today by vendors with appropriate access on existing hardware. Next, we explain that Titrate's hardware requirements are on-par or lower than other approaches that have been implemented or considered practical.

Titrate needs access to the minimum queue length, which is similar to CoDel's minimum packet sojourn time, and to zero queue duration, which is accessible *e.g.,* by setting the high and low thresholds of LANZ [2] (Broadcom backend) to 0. Titrate updates thresholds only every few hundred milliseconds (*i.e.,* once per monitoring interval), far less frequently than DT and ABM, which update on every packet arrival. Titrate also computes a modified moving average of queue lengths, which is already used by AQMs *e.g.,* RED [21]. Critically, because Titrate is queue-local, it is easier to implement than buffer management schemes *e.g.,* ABM [8] or even DT [17]: Titrate does not require access to the device-global unoccupied buffer size (needed by both DT and ABM) or to the number of active queues of the same priority across all ports (needed by ABM).

## 5 Simulator Evaluation

We evaluate Titrate in packet-level simulation with ns-3.34 [4]. We compare with state-of-the-art AQMs that operate on a single queue, CoDel [26], COBALT [35] and PIE [36], for queue-level benefits, and status-quo buffer management scheme DT [17] for device-level benefits. We show that: Titrate achieves high throughput, low latency and effective burst absorption for diverse queue compositions (§5.2). Titrate offers device-wide benefits: by reducing buffer usage per port, it frees shared memory so multiple ports on the device can achieve better performance under load (§5.3). Titrate adapts to dynamic traffic and network conditions swiftly and efficiently (§5.4).

### 5.1 Methodology

**Topology.** We use a star topology where multiple sending and receiving servers are connected to one switch. The exact numbers of senders and receivers will be specified in each experiment. Each sender sends one flow. We simulate an output-queued switch that is common in practice [7, 40]. Unless otherwise specified, the device is asymmetric with 2Gbps bandwidth on input ports and 1Gbps bandwidth on output ports. Each port has one queue. Unless otherwise specified, we provision 2*BDP[4] buffer memory on switch. Links from the switch to the receivers have 1ms propagation delay. Links from the senders to the switch mimic real-world propagation delays and will be specified in each experiment.

**Simulation.** We use a monitoring interval of length 500ms.

[4]Port bandwidth multiplied by the average RTT of flows in the queue.

We set $c_{dec} = 1$, $c_{inc} = 1$, $c_{thresh} = 3$. The simulation runs for 200s unless otherwise specified.

**Workload.** We experiment with long-lived flows and bursty flows. We vary CCAs for long-lived flows to ensure realistic and diverse traffic composition, including Cubic [25], BBR (version 1) [16] and Mix (a mixture of Cubic, BBR, Yeah [14], Illinois [29], Vegas [15], Htcp [28], Bic [41], Reno [22] and Scalable [27] according to the ratio reported in the survey paper [33]). The flow sizes are set to 10GB and the flows effectively keep sending throughout the simulation. We generate bursty flows from a dataset of Web traffic running under Cubic, following prior work [32]. The dataset was generated using selenium [5] to automatically load the Alexa Top-1000 websites [1] through Google Chrome in November 2023. The burstiness mostly comes from loading different objects on the webpage.

**Baselines.** We compare Titrate with CoDel [26], COBALT [35], PIE [36], a static threshold at 1*BDP (Static) for queue-level experiments, and with DT [17] for device-level experiments. We configure CoDel and PIE with the RFC-recommended defaults [34, 37], and configure COBALT with ns-3's default parameters. We use $\alpha = 1$ for DT following Arista [8].

## 5.2 Performance across Queue Compositions

**Titrate ensures high throughput and low latency for a wide range of queues.** We experiment with nine queue compositions that differ in RTT, CCA and number of flows: flows in the queue are of 50ms (SmallBDP) or 300ms (LargeBDP) RTT; the CCAs of the flows are Cubic only, BBR only or Mix; the number of flows is randomly selected from a range and the range is 10 to 50 for (S) and 500 to 1000 otherwise. For each composition, we randomly generate five queues and report their average throughput and queuing latency with an error bar showing the minimum and maximum in Figure 8. Across a wide range of queue compositions, Titrate achieves high throughput and low latency simultaneously, outperforming CoDel, COBALT, PIE and Static. In particular, the minimum throughput of Titrate is 38.49%, 32.21% and 13.62% larger than that of CoDel, COBALT and PIE. Titrate has a comparable throughput with Static (about 0.1% difference on average) but incurs 58.66% less queuing latency. CoDel, COBALT and PIE make simplistic assumptions on queue dynamics and thus are unable to effectively control queue growth while keeping high throughput across diverse queue compositions. Static uses as much buffer memory as possible, thus ensuring the highest possible throughput but sacrificing queuing latency.

We also reproduce the canonical queue composition for CoDel, COBALT and PIE: a queue with only a small number of Cubic flows of an average 50ms RTTs and summarize our results in Figure 9. As expected, CoDel, COBALT and PIE perform very well. Notably, Titrate too performs well in this case with an equally high throughput and a queuing latency higher than CoDel and COBALT but lower than PIE.

**Titrate effectively absorbs bursts.** We experiment with 20 randomly selected web traces (*i.e.,* bursts) from the Web traffic dataset, which have variable burst size and rate. The experiment is run with background traffic of Mix flows of 50ms average RTT. The number of background flows is randomly selected between 500 and 1000. Figure 10 shows the CDF of burst completion times and the number of drops in the bursts. Titrate reduces the average burst completion times by 3.36%, 57.75%, 36.86% and 27.51% compared to CoDel, COBALT, PIE and Static, respectively. This highlights the effectiveness of Titrate in absorbing bursts. In addition, compared to Titrate (No EQlen), Titrate reduces the average burst completion times by 58.98%. This verifies the effectiveness of EQlen in allowing bursts to escape thresholding and enqueue into the buffer.

**Titrate is resilient to marking errors.** To compare with L4S, we experiment with two scenarios running 50ms-RTT flows. In (S), the queue has a small number of flows (randomly between 10 and 50 flows); in (L), the queue has a large number of flows (randomly between 500 and 1000 flows). To enable L4S in ns-3, we run DCTCP under "L4S service" and CUBIC under "Classic service". In Titrate, both types of flows go to the same queue, while in the L4S scheme, flows go to two queues according to the dualQ scheme. Figure 11 reports the throughput and average queuing latency over five random trials for three schemes: Titrate, L4S with correct markings, and L4S with *only one single mismarked flow i.e.,* one Cubic flow is incorrectly marked to be in the "L4S service". With correct markings, L4S unsurprisingly achieves high throughput and low latency, with Cubic flows contributing almost all latencies. However, introducing just one mismarked Cubic flow into the L4S class inflates the queuing delay to 1.86 times and 7.13 times higher than Titrate in the two scenarios, respectively. In contrast, Titrate does not rely on marking and therefore is naturally resilient to any (intentional or unintentional) marking errors at end hosts.

**Titrate's performance scales to higher bandwidth.** In Figure 12, we experiment with a queue of 10 10ms-RTT Cubic flows with 1Gbps links and 10Gbps links. Figure 12a shows that with both 1Gbps and 10Gbps links, Titrate achieves full throughput with low latency, and the latencies are 13.83% and 34.07% lower than CoDel, the strongest baseline from previous results. Figure 12b further demonstrates how threshold in Titrate effectively adjusts and reaches the minimum for high throughput and low latency.

## 5.3 Device-wide Benefits

**Titrate improves throughput for all ports on a device with small buffer.** In Figure 13, we simulate four devices with 2,3,4,5 output ports and 11MB,12MB,13MB,14MB total buffer memory respectively. Each output port has a single queue. On each device, port 1 has a queue with 50 Cubic flows of 300ms RTT on average (longRTT) and all other ports have queues with 50 Cubic flows of 50ms RTT on average (shortRTT). We report the average throughput and queue length for the longRTT queue and across all shortRTT queues on device.

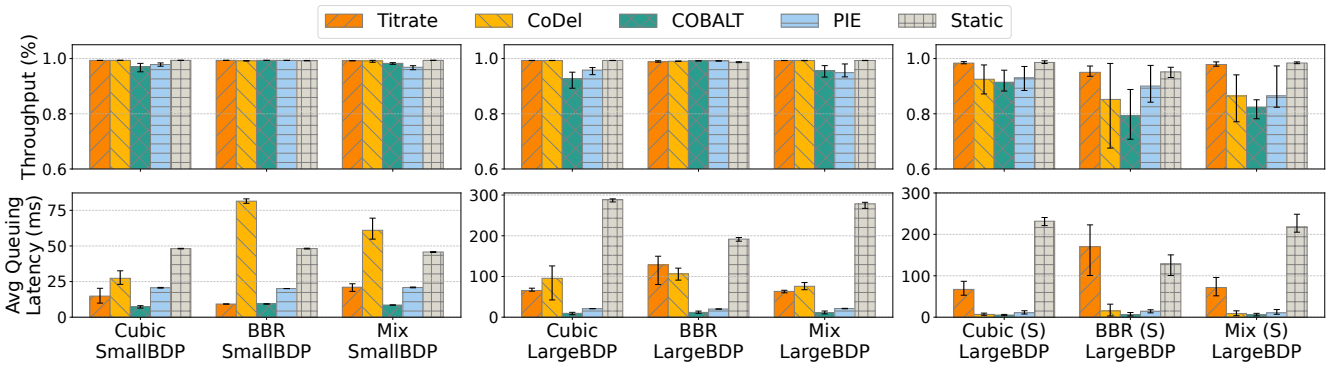



Figure 8: Titrate simultaneously achieves high throughput and low latency across diverse traffic compositions. CoDel, COBALT and PIE control queuing delay but at the expense of throughput. Static thresholds ensure high throughput, but at the expense of latency.

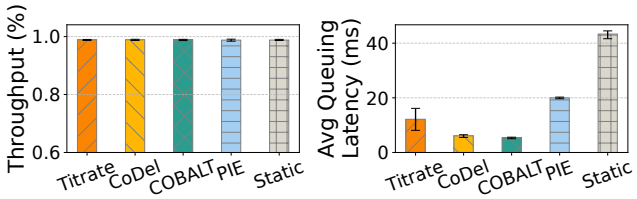



Figure 9: CoDel, COBALT and PIE perform well for the queue composition they were optimized for *i.e.,* few Cubic flows of short RTTs.

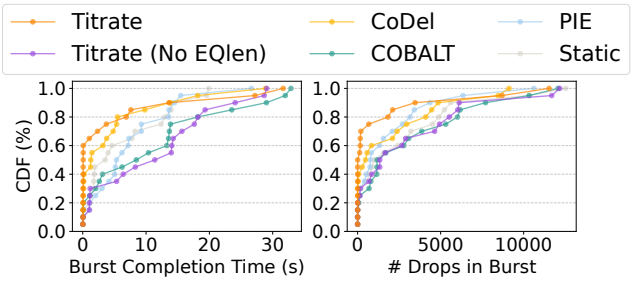



Figure 10: By treating bursts as queue-length outliers with EQlen, Titrate achieves lower burst completion times on average.

For shortRTT queues, both Titrate and DT achieve full throughput but Titrate uses 68.13% less buffer memory. As a result, the longRTT queue in Titrate can use more buffer and thus achieve a high throughput of 98.07% on average. In comparison, in DT, all queues have identical thresholds, and the longRTT queue can only achieve 92.21% throughput on average. A more concerning observation is that with DT the longRTT queue is even shorter than the shortRTT queues because of its larger drop rate. In addition, as the number of shortRTT queues increases and the buffer contention on device intensifies, the throughput of LongRTT queues further degrades. As queue compositions are dynamic, statically changing DT's configuration to reduce buffer usage of specific queues (same as statically configuring thresholds), is risky for performance. This longRTT vs shortRTT experiment illustrates Titrate 's advantage on shared-buffer switches: by tailoring thresholds to each queue's needs, Titrate allocates

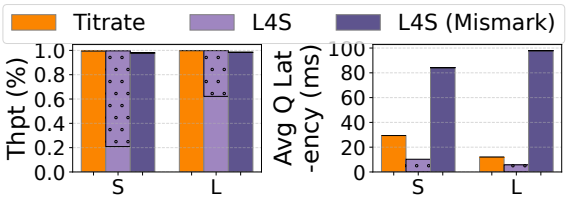
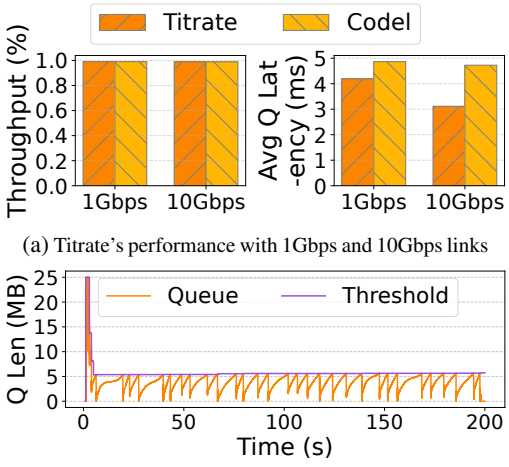



Figure 11: L4S with correct markings achieves high throughput and low latency. However, *just one* mismarked flow risks inflating queuing latency. Titrate does not bear this risk because it does not rely on marking. Bars with dots are contributions from the non-L4S-enabled flows. S and L represent two queue compositions that differ in number of flows.




(a) Titrate's performance with 1Gbps and 10Gbps links

(b) Queue length and threshold variations at 10Gbps.

Figure 12: Titrate outperforms CoDel (the strongest baseline) with both 1Gbps and 10Gbps links, demonstrating that Titrate keeps its performance advantage with higher-bandwidth links.

buffer where it is most effective, which is especially valuable on small-buffer devices with buffer contention. The same benefit extends to other heterogeneous queue mixes.

**Titrate improves burst completion time on other ports sharing the device.** In Figure 14, we simulate four devices with 12MB,16MB,20MB,24MB total buffer memory, respec-

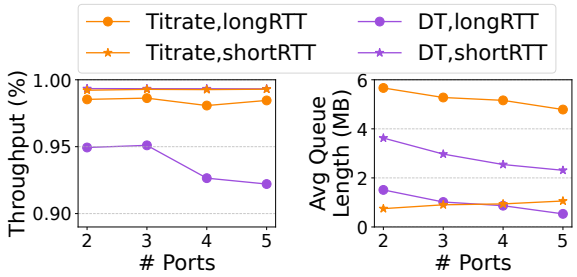



Figure 13: Splitting buffer equally across all queues on a device is not efficient when the queues are of distinct compositions. Titrate keeps per-queue buffer usage to the minimum, saving shared buffer memory for queues that actually need it (*e.g.,* those with longer RTT), hence outperforming DT.

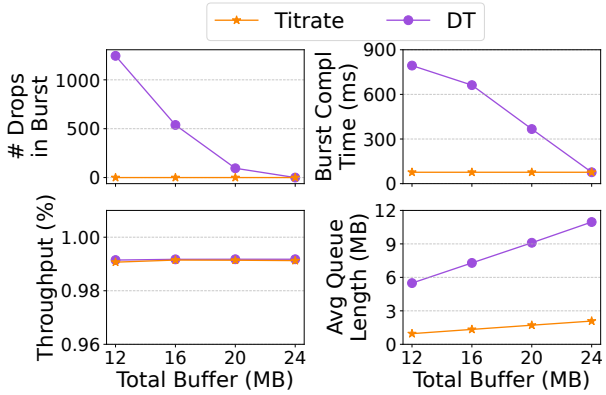



Figure 14: Titrate keeps queues to the minimum buffer needed for full throughput, and thus leaves more space in the shared buffer for bursts on other ports of the same device.

tively. Each device has two ports and a single queue per port. Port 1 has 50 Cubic flows of 50ms average RTT. 20 seconds into the simulation, Port 2 sees 10 consecutive bursts, drawn randomly from the Web traffic dataset, with Poisson arrival of rate 0.1. Across all devices, Titrate drops no packet and thus achieves the smallest possible burst completion time of 76.53ms. In comparison, DT achieves a burst completion time of 76.53ms with 24MB buffer memory, but this jumps to a staggering 793.70ms with 12MB – more than 10 times higher! This is because both Titrate and DT achieve full throughput in port 1, but Titrate uses 81.47% less buffer than DT on average across devices. This way, Titrate leaves more room for flows actually in need of more buffer space – in this case, the bursts on port 2. This again highlights the importance of Titrate's ability to cater to individual queue needs, especially when working with small-buffer devices.

## 5.4 Adaptability to Dynamic Network & Traffic

**Titrate adapts to changing available bandwidth swiftly and efficiently.** Available bandwidth to a queue can change dynamically in the Internet. Figure 19 in Appendix 8.4 plots the variation in queue lengths over time when the available bandwidth to a queue with 500 Mix flows of an average RTT

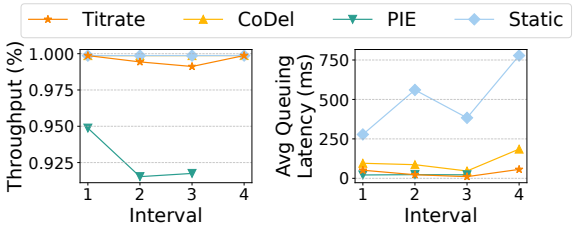



Figure 15: Titrate adapts to changing available bandwidth swiftly and efficiently.

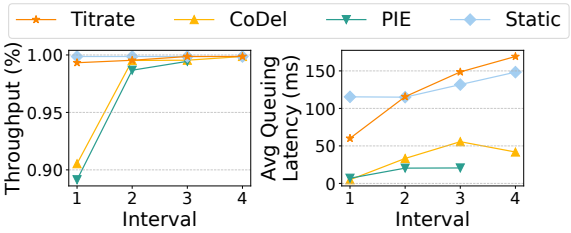



Figure 16: Titrate achieves high throughput and low latency with dynamic traffic.

of 300ms changes. Titrate adjusts its queue threshold swiftly with the changing network conditions, decreasing threshold when it sees an opportunity to do so while maintaining a high throughput across bandwidth changes. Figure 15 reports the average throughput and queuing latency for the four intervals with different bandwidths. Both Titrate and CoDel achieve high throughput and low latency, while PIE incurs throughput loss and Static incurs excessive queuing delay.

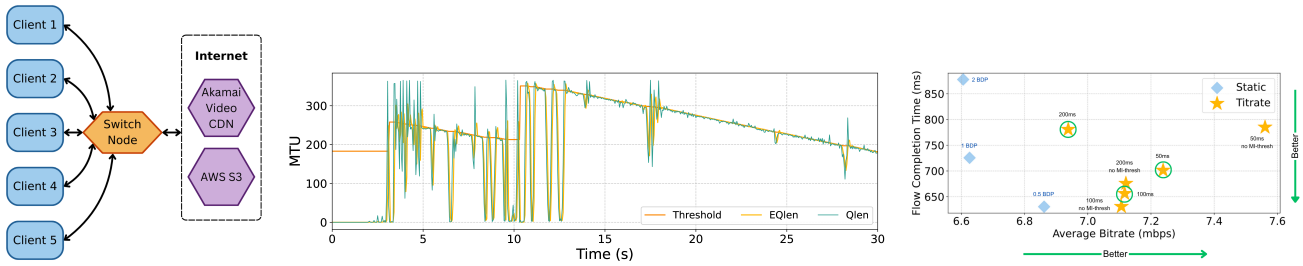
**Titrate achieves high throughput and low latency with dynamic traffic.** We experiment with dynamic traffic. The simulation runs for 200s and for every 50s, we inject flows of twice the amount of existing flows and of a random RTT and Cubic-to-BBR ratio. The exact traffic changes are specified in Figure 20 in Appendix 8.4, which also plots the queue length variations over time and shows that Titrate adapts efficiently to dynamic traffic, achieving 99.65% throughput overall. Figure 16 reports the average throughput and queuing latency for the four intervals with different traffic. In intervals 2 & 3, we have more BBR flows which need more buffer, and Titrate increases the queue threshold and queue length promptly. In interval 4, the queue's buffer need decreases, and Titrate enters additive-decrease. Both CoDel and PIE incur about 10% throughput loss in interval 1, as they fail to adapt to the traffic condition.

# 6 Internet Testbed Evaluation

In this section, we validate Titrate in a more realistic Internet setting and against more realistic traffic, including video. We demonstrate that Titrate outperforms various static thresholds, is not sensitive to parameterization and is resilient to changes in traffic composition.

## 6.1 Methodology

**Topology.** We set up the topology shown in Fig. 17a. Five client nodes connect to the Internet via a single switch node within




(a) (b) (c)

Figure 17: **(a)** Our CloudLab testbed topology; **(b)** Titrate in action while 4 clients streaming video and 1 client downloading small files. During the first 15 seconds, all 4 video clients initialize simultaneously, causing Titrate to increase the threshold fast. Once it achieves high throughput, Titrate cautiously reduces the threshold to minimize queuing latency; **(c)** Across parameterizations, Titrate outperforms multiple different static thresholds. Each data point represents the average metrics for a given parameterization of Titrate or static. Circled stars indicate full Titrate (MI-thresh enabled).

the same CloudLab site in Utah. The queue management logic in the switch (static or Titrate) is implemented in eBPF with eXpress Data Path (XDP) packet processing. We test various configuration of static threshold and Titrate. We use the Linux *tc* command to control RTT between the clients and switch.

**Workload.** Four out of the five clients stream a video using headless Chromium version 128.0.6613.84 with Python selenium, streaming the sample "Big Buck Bunny" video hosted on Akamai by the Dash Industry Forum [23]. The player is the DashJS sample client with the default configuration. Because the clients are using DASH, the bitrate (quality) of the video can change dynamically over time depending on the client's prediction of network performance and congestion. The fifth client requests data sequentially from AWS S3 ranging from 0.1 megabits to 1 megabit, very small file sizes compared to the amount of data the video streams transmit (the maximum quality of the a single video stream targets in our experiments is 12 megabits per second). The other 4 clients stream a video identically to the previous experiment. All experiments are averaged over 5, 2-minute long trials.

## 6.2 Results

**Titrate adapts thresholds as expected.** Figure 17b shows Titrate in action for a sampled time window of our experiment. Titrate increases the threshold (measured in MTU) rapidly to achieve full throughput, which is threatened as new clients start watching a video in the first 15 seconds. When Titrate reaches full throughput, it steadily decrease the threshold to reduce queuing latency.

**Titrate achieves a better tradeoff between bit-rate and flow-completion-time compared to static.** Figure 17c shows the performance *i.e.,* tradeoff between bit rate and flow-completion-times of various parameterizations of the static buffer (0.5 BDP, 1 BDP, 2 BDP) and Titrate– variations are in the monitoring interval length (50ms, 100ms, 200ms). To get more insights we also run Titrate with MI-thresh off. Each data point is one parameterization of Titrate or static. Titrate outperforms the static threshold, even when the value of BDP is known in advance (flows have a consistent RTT with little variance), a setting that already favors static. Critically, Titrate is not highly sensitive to its parameterization, ensuring that a real-world deployment, even if not perfectly configured, will maintain good performance. Notably, even the worst-performing parameterization of Titrate presents a better tradeoff between bit rate and completion time.

**Titrate is resilient to changes in traffic composition, specifically in RTT.** One of the main limitations of static buffer thresholds is their inability to handle changing traffic compositions. Even if a static threshold is ideal for one scenario, Internet traffic constantly evolves, making any static threshold suboptimal in most situations. Titrate, by contrast, makes no assumptions about flow properties, enabling it to better handle variations in traffic composition. To test this, we compare Titrate and static thresholds when the RTT of the video streams and the short flows increase by just 20ms.Averaged across the parameterizations previously discussed, Titrate increases from 686ms to 817ms average flow completion time (19.8% increase), while static increases from 726ms to 962ms (32.6% increase). While increasing RTT would typically increase flow completion time, Titrate adapts to minimize the impact of this RTT increase.

## 7 Conclusion

This paper presents Titrate, a principled and practical solution to the longstanding challenge of queue management in Internet routers. Titrate adapts thresholds based on live queue dynamics and is shown to consistently achieve high throughput and low latency in simulation and testbed.

## Acknowledgements

We sincerely thank our shepherd, Vishal Misra, and the anonymous reviewers for their valuable feedback. We also thank Stefan Schmid for helpful discussions on an earlier version of the paper. This work was supported by the National Science Foundation (NSF) through Grants CNS-2442625, and CNS-231944.

## References


[1] Alexa top websites » expireddomains.net. https://member.expireddomains.net/domains/researchalexamillion/.

[2] Arista lanz overview. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_LANZ_Overview_TechBulletin_0213.pdf.

[3] Netflix vs. comcast – the peering problem. https://www.vyprvpn.com/blog/post/netflix-vs-comcast-the-peering-problem.

[4] Ns3 network simulator. https://www.nsnam.org/.

[5] Selenium. https://www.selenium.dev/.

[6] Traffic management user guide (qfx series switches and ex4600 switches). https://www.juniper.net/documentation/us/en/software/junos/traffic-mgmt-qfx/topics/example/cos-shared-buffer-allocation-lossy-ethernet-pause-qfx-series-configuring.html?utm_source=chatgpt.com.

[7] Understand queue buffer allocation on catalyst 9000 switches. https://www.cisco.com/c/en/us/support/docs/switches/catalyst-9500-series-switches/218444-understand-queue-buffer-allocation-on-ca.html.

[8] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: Active buffer management in datacenters. In *Proceedings of ACM SIGCOMM*, 2022.

[9] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. Reverie: Low pass filter-based switch buffer sharing for datacenters with rdma and tcp traffic. In *21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), Santa Clara, CA*, 2024.

[10] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[11] Maria Apostolaki, Laurent Vanbever, and Manya Ghobadi. Fab: Toward flow-aware buffer sharing on programmable switches. In *ACM Workshop on Buffer Sizing*, 2019.

[12] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. *Sizing router buffers*, volume 34. ACM, 2004.

[13] M Bagnulo and G White. Low latency, low loss, and scalable throughput (l4s) internet service: Architecture. 2023.

[14] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[15] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[16] Neal Cardwell, Yuchung Cheng, Soheil Yeganeh, and Van Jacobson. Bbr congestion control. *Working Draft, IETF Secretariat, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00*, 2017.

[17] Abhijit K Choudhury and Ellen L Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions On Networking*, 6(2):130–140, 1998.

[18] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. Rfc 6928: Increasing tcp's initial window, 2013.

[19] Sujal Das and Rochan Sankar. Broadcom smart-buffer technology in data center switches for cost-effective performance scaling of cloud applications. *Broadcom White Paper*, 2012.

[20] Koen De Schepper, Olga Bondarenko, Ing-Jyh Tsang, and Bob Briscoe. Pi2: A linearized aqm for both classic and scalable tcp. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 105–119, 2016.

[21] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug 1993.

[22] Sally Floyd, Tom Henderson, and Andrei Gurtov. Rfc3782: The newreno modification to tcp's fast recovery algorithm, 2004.

[23] Dash Industry Forum. dash.js. https://github.com/Dash-Industry-Forum/dash.js.

[24] Yashar Ganjali and Nick McKeown. Update on buffer sizing in internet routers. *ACM SIGCOMM Computer Communication Review*, 36(5):67–70, 2006.

[25] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[26] V Jacobson and N Kathleen. Controlling queue delay-a modern aqm is just one piece of the solution to bufferbloat. *Asscociation for Computing Machinery (ACM Queue)*, 2012.

[27] Tom Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.

[28] Douglas Leith, R Shorten, and Y Lee. H-tcp: A framework for congestion control in high-speed and long-distance networks. In *PFLDnet Workshop*, 2005.

[29] Shao Liu, Tamer Başar, and Ravi Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, pages 55–es, 2006.

[30] Matt Mathis and Andrew McGregor. Buffer sizing: a position paper.

[31] Nick McKeown, Guido Appenzeller, and Isaac Keslassy. Sizing router buffers (redux). *ACM SIGCOMM Computer Communication Review*, 49(5):69–74, 2019.

[32] Zili Meng, Nirav Atre, Mingwei Xu, Justine Sherry, and Maria Apostolaki. Confucius: Achieving consistent low latency with practical queue management for real-time communications. *arXiv preprint arXiv:2310.18030*, 2024.

[33] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The great internet tcp congestion control census. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, pages 59–60, 2020.

[34] Kathleen Nichols and Van Jacobson. Rfc 8289: Controlled delay active queue management, 2018.

[35] Jendaipou Palmei, Shefali Gupta, Pasquale Imputato, Jonathan Morton, Mohit P Tahiliani, Stefano Avallone, and Dave Täht. Design and evaluation of cobalt queue discipline. In *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6. IEEE, 2019.

[36] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pages 148–155, July 2013.

[37] Rong Pan, Natarajan Preethi, Fred Baker, and Greg White. Rfc 8033: Proportional integral controller enhanced (pie), 2017.

[38] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 685–699, 2020.

[39] Bruce Spang, Serhat Arslan, and Nick McKeown. Updating the theory of buffer sizing. *ACM SIGMETRICS Performance Evaluation Review*, 49(3):55–56, 2022.

[40] Tim Stevenson. Nexus 9000 architecture, 2020.

[41] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524. IEEE, 2004.

# 8 Appendix

## 8.1 Limitations of COBALT

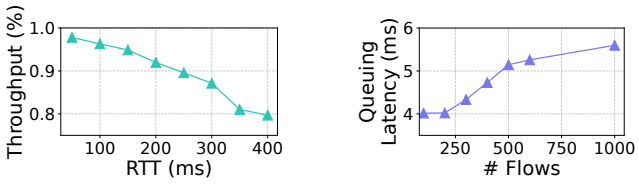



Figure 18: COBALT still suffers from severe throughput loss when the average flow RTT in the queue increases (Left) and reduces queuing latency inflation when the number of flows in the queue increases (Right).

## 8.2 Discussion on Signals to Monitor

| Metrics | Signals | Requirements | | |
|---|---|---|---|---|
| | | FG | BI | QI |
| Thpt | sum(pkts) | ✓ | ✓ | |
| | sum(bytes) | ✓ | ✓ | |
| | min(qlen) | | ✓ | ✓ |
| | zeroqd | ✓ | ✓ | ✓ |
| Latency | avg(qlen) | ✓ | | |
| | avg(qlat) | ✓ | ✓ | |
| | max(qlen) | ✓ | | |
| | max(qlat) | ✓ | ✓ | |
| | min(qlen) | ✓ | ✓ | ✓ |
| | min(qlat) | ✓ | ✓ | ✓ |

Table 1: Signals for throughput and latency. FG stands for fine-grained, BI stands for bandwidth-independent, and QI stands for queue-independent.

Table 1 presents the signal candidates and how they meet or fail our requirements. For example, minimum queue length (*i.e.,* min(qlen)) is too coarse-grained to be a good signal for throughput as it indicates whether we have throughput loss or not but not how severe the loss is. Packet count (*i.e.,* sum(pkts)) has a variable target depending on bandwidth, thus failing to be a good signal for throughput. Similarly, average queue length/latency (*i.e.,* avg(qlen)/avg(qlat)) has a variable target depending on queue compositions, thus failing to be a good signal for latency. Note that both minimum queue length and latency (*i.e.,* min(qlen) and min(qlat)) suffice as a signal for latency and Titrate is indifferent to which one to use.

## 8.3 Discussion on Design Parameters

We discuss how to set the four parameters in Titrate.

**Monitoring interval length.** A small interval collects less information on the severity of throughput loss and aggregates over fewer data points for a less reliable minq signal; a large interval can be easily contaminated by noises and prolong sub-optimal performance *e.g.,* throughput loss.

**Decrease/increase step size ($c_{dec}$,$c_{inc}$).** The two parameters control how quickly we converge to the ideal threshold. A larger decrease step size and a smaller increase step size keeps queuing latency low but can be too aggressive and lead to throughput loss. On the other hand, a smaller decrease step size and a larger increase step size keeps throughput high but can be too conservative and lead to excessive queuing delay.

**MI-thresh multiplier ($c_{thresh}$).** A small multiplier is aggressive about where the safe threshold is and can lead to throughput loss later when decreasing too fast; a large multiplier is more conservative but essentially does not speed up the additive decrease.

## 8.4 Time Series for Adaptability to Dynamic Network & Traffic

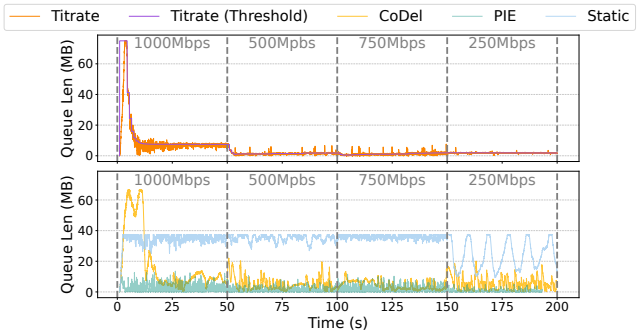



Figure 19: Titrate adapts to changing available bandwidth swiftly and efficiently.

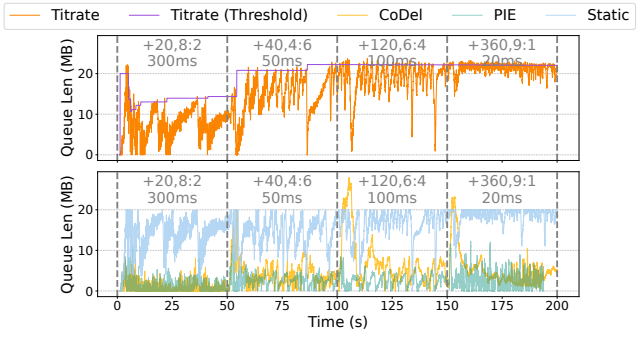



Figure 20: Titrate achieves high throughput and low latency with dynamic traffic.