A Cross-Layer Congestion Control Mechanism for Shuffle-Intensive Analytics with Deadline-Aware Rate Adaptation

Li Wei[1] and Zhou Ming[2]

[1] Liaoning University of Technology, Xueyuan Street, Jinzhou 121001, Liaoning, China
[2] Henan University of Science and Technology, Kaiyuan Avenue, Luolong District, Luoyang 471023, Henan, China

Abstract

Shuffle-intensive analytics frameworks commonly transform computation into alternating phases of local processing and all-to-all data exchange. In modern clusters, the shuffle phase frequently dominates both completion time and tail variability because it induces bursty many-to-many transfers, transient incast at reducers, and queue buildup along shared fabric bottlenecks. At the same time, interactive and production analytics are often executed under job- or stage-level deadlines, where the relevant objective is not only throughput but also timely delivery of intermediate partitions so that downstream tasks can begin without prolonged blocking. This text describes a cross-layer congestion control mechanism tailored to shuffle traffic with explicit deadline-aware rate adaptation. The mechanism couples application-layer knowledge of shuffle structure and remaining bytes with transport-layer pacing and explicit congestion feedback, allowing rates to be shaped by urgency while still respecting network stability. A coflow-oriented abstraction is used to associate multiple concurrent TCP-like flows with a single shuffle stage, exposing per-reducer and per-stage completion requirements. A lightweight price signal derived from ECN-marked congestion and receiver-side service pressure steers host rate controllers toward allocations that reduce queue occupancy and limit incast collapse. Deadline awareness is implemented by an urgency-weighted control law that increases rate when slack diminishes, but caps the response when congestion prices rise, yielding bounded queues and predictable fabric utilization. The overall design emphasizes deployability in commodity stacks through minimal switch features, kernel-level pacing hooks, and analytics runtime integration.

1 Introduction

Cluster analytics engines execute directed acyclic graphs of stages where each stage contains a set of tasks, and tasks exchange intermediate data through shuffles [1]. A shuffle generally materializes as many-to-many communication: each mapper emits partitions for many reducers, and each reducer pulls partitions from many mappers. This pattern differs from long-lived elephant flows and also differs from single-sender incast microbenchmarks, because shuffle transfers are short to medium in duration, fan-in and fan-out are large, and the set of active connections changes rapidly as tasks finish and retries occur. Even when aggregate bytes are moderate, the temporal alignment of transfers can be severe, particularly when many mappers complete near-synchronously and reducers begin pulling at similar times. The resulting incast can drive queue spikes and packet loss, which conventional loss-based congestion control interprets only after overshoot, amplifying completion-time variance [2].

A second complication is that analytics pipelines are increasingly deadline-driven. Deadlines appear as interactive query latency targets, scheduled batch windows, service-level objectives for ETL pipelines, and internal cluster scheduling constraints that aim to keep downstream resources utilized. In a shuffle stage, the practical deadline is often a stage barrier: reducers cannot complete until they have received all required partitions, and downstream stages cannot begin until upstream barriers are satisfied or until enough data is available to pipeline execution. Consequently, meeting a job deadline depends on the timely completion of a set of coupled transfers rather than on the average throughput of any single flow. The same job may tolerate slower delivery of some partitions while requiring fast delivery of others when the critical path runs through specific reducers or straggling mappers [3]. Conventional transport stacks treat each connection independently and are intentionally agnostic to higher-layer deadlines, making them ill-suited to exploit such structure.

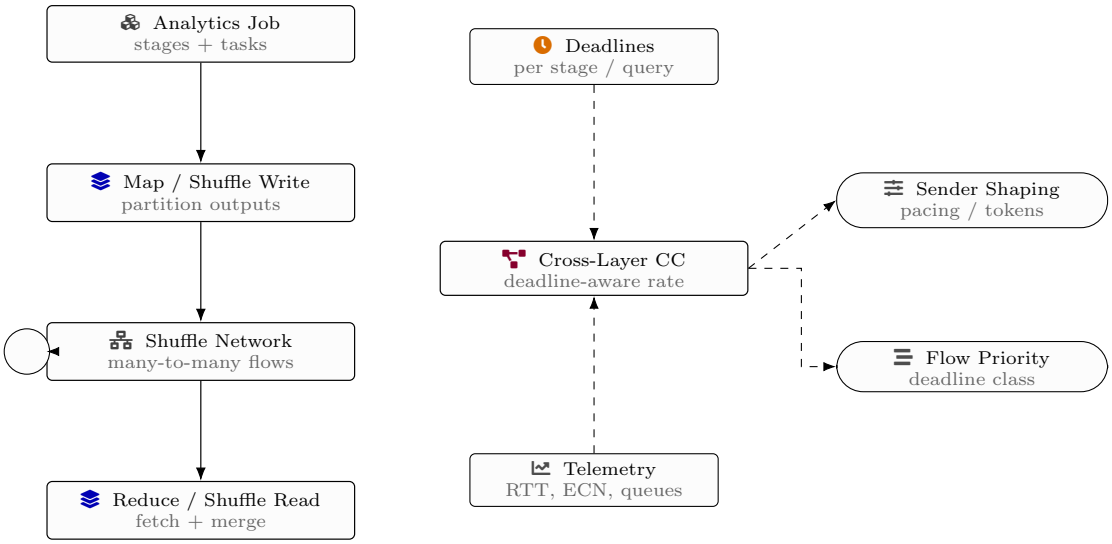

Figure 1: System overview for shuffle-intensive analytics: a cross-layer congestion controller consumes stage deadlines and network/transport telemetry to shape sender pacing and prioritize shuffle flows, reducing deadline misses under incast-heavy traffic.

Table 1: Data-Center and Shuffle Configuration

Parameter	Value	Scope	Notes
Servers	256	Cluster	16 racks, 16 servers per rack
Link capacity	40 Gbps	Host–ToR	Full-duplex Ethernet
Baseline RTT	75 μs	Path	No queuing, same rack
Shuffle fan-in	20–200 tasks	Job	Map-heavy analytic pipelines
Shuffle fraction	60–80% bytes	Job	Dominant in completion time

The congestion-control question for shuffle traffic is therefore not only how to share capacity fairly but also how to apportion capacity across coflows so that stage completion aligns with deadlines. Any mechanism that pushes urgency directly into rates risks destabilizing the network if it ignores shared bottlenecks, and any mechanism that focuses purely on fabric stability risks violating deadlines by allocating capacity too conservatively to time-critical transfers. A practical design must also work within operational constraints: switches may support ECN but not complex scheduling; hosts must handle large numbers of connections; analytics runtimes should integrate without rewriting the entire shuffle stack; and rate updates must remain stable under milliseconds-scale RTTs.

This text develops a cross-layer mechanism that combines three information sources [4]. First, the application layer supplies structure: which flows belong to which shuffle stage, how many bytes remain to be sent to each reducer, and the stage-level deadline or target completion time. Second, the transport layer supplies fine-grained pacing and congestion feedback, using ECN marking and, when available, receiver service-pressure signals that reflect how quickly reducers can drain and process incoming data. Third, a control layer within the host computes per-coflow target rates from an urgency-weighted allocation that accounts for both remaining slack to deadline and current congestion prices. The objective is not to eliminate all deadline misses, which may be impossible under overload, but to reduce miss ratio and reduce tail completion times by steering scarce capacity toward critical transfers while preserving bounded queues.

A key design choice is to treat shuffle transfers as coflows: a coflow is a set of related flows that jointly determine a stage's readiness [5]. For a reducer, the coflow completion time depends on the slowest incoming partition among many senders, and for a stage the completion time depends on the slowest reducer. Per-flow fairness can therefore be misaligned with stage objectives: allocating equal bandwidth to all connections can leave critical reducers waiting for the last few partitions, while non-critical flows consume capacity. Conversely, aggressively prioritizing a subset of flows can create starvation and instability. The mechanism presented here embeds deadline signals in coflow-level weights and converts them into rates through a price-based controller whose stability properties resemble those of congestion pricing in networks.

The remainder of the text proceeds as follows [6]. The next section describes shuffle communication characteristics and the motivation for cross-layer coupling. The problem formulation section defines the network model, the coflow abstraction, and deadline semantics, and introduces an optimization viewpoint. The design section describes the cross-layer congestion control architecture, including host and switch components and the coflow state main-

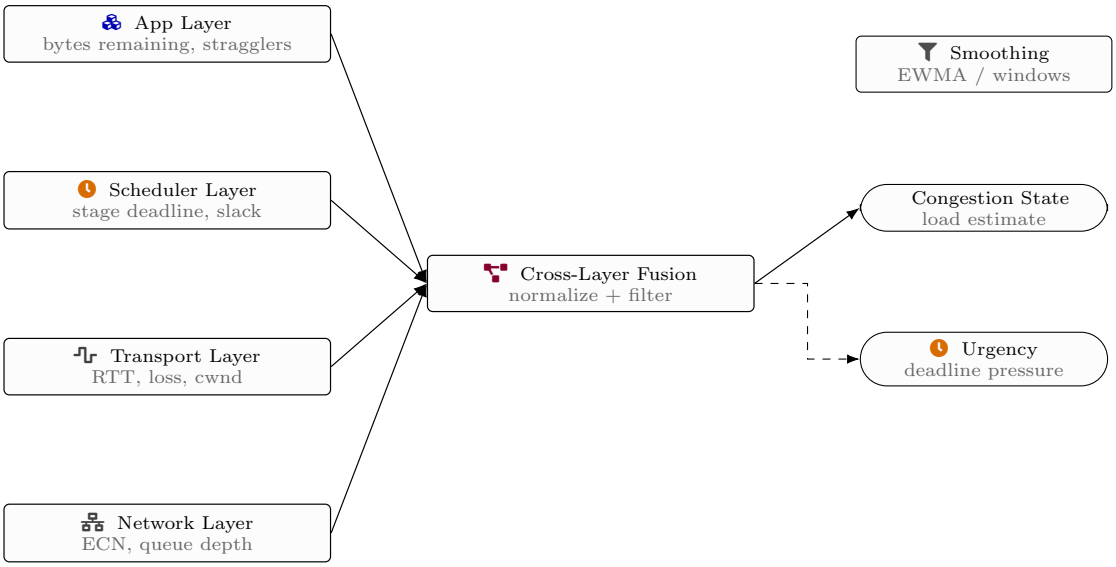

Figure 2: Cross-layer signal interface: application and scheduler hints (remaining bytes, deadline slack) are fused with transport and network feedback (RTT, loss, ECN/queues) to estimate congestion and urgency for rate decisions.

Table 2: Cross-Layer State Exposed to the Rate Controller

Layer	Variable	Symbol	Update period
Application	Job deadline	D_j	On submission
Runtime	Remaining shuffle bytes	B_f	Every 5 ms
Transport	Smoothed RTT	R_f	Every ACK
Network	ECN mark probability	p_f	Per congestion epoch
Host	Per-core CPU utilization	u_h	Every 10 ms

tained in the runtime. The deadline-aware adaptation section derives an urgency-weighted rate law and discusses stability and corner cases such as overload and task retries. The implementation and evaluation section discusses deployable integration points and a methodology for assessing the design under representative shuffle workloads [7]. The conclusion summarizes the central mechanism and outlines constraints that shape its applicability.

2 Background and Motivation

Shuffle traffic inherits multiple sources of burstiness. At the application layer, map tasks often finish in waves because they process similarly sized input splits under homogeneous resource allocations. At stage boundaries, reducers are typically scheduled together, and they begin pulling output partitions after map output becomes available or after a threshold is met. At the transport layer, each reducer may open hundreds or thousands of concurrent connections, and each mapper may serve partitions to many reducers [8]. Even when each individual connection is moderate in size, the aggregate offered load into a reducer's receiving port can exceed line rate by a large factor during synchronized pulls. When loss-based congestion control is used, packet loss is triggered at the switch queues, and multiple senders reduce their windows after RTT-scale delays. The transient before convergence can be long relative to short shuffle flows, leading to completion-time sensitivity to initial bursts rather than steady-state.

Datacenter fabrics are commonly engineered for high bisection bandwidth but not necessarily for worst-case all-to-all with strict tail guarantees. Oversubscription at top-of-rack or aggregation layers means that a rack-local hotspot can spill into shared uplinks [9]. ECN-capable switches can mark packets when queues exceed a threshold, enabling end-host algorithms such as DCTCP-like controllers to maintain low queues by responding to mark fraction instead of loss. However, ECN feedback alone does not encode which transfers are time-critical. Without deadline awareness, an ECN-based controller aims at queue stability and approximate fairness, leaving the scheduling of urgency to the application layer. In shuffle workloads, the application scheduler often lacks direct control over network pacing; it can delay task launches or throttle at coarse granularity, but the fine-grained interactions among thousands of connections remain governed by the transport.

Deadline semantics in analytics are also subtle [10]. A "deadline" may be a strict end-to-end completion time,

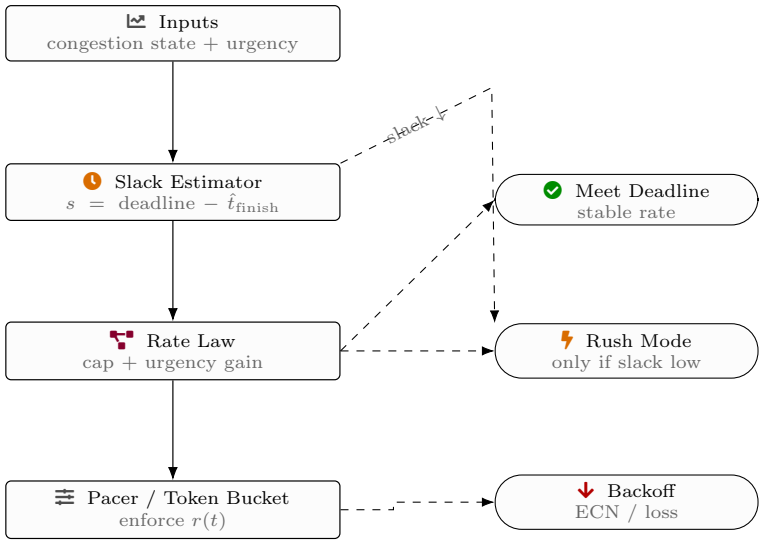

Figure 3: Deadline-aware rate adaptation: slack estimation predicts whether shuffle transfers can finish before the stage deadline, and a rate law adjusts sender pacing—favoring stability when slack is ample and briefly increasing rate only under low slack while respecting congestion feedback.

Table 3: Deadline Classes and Urgency Scaling

Class	Deadline range	Urgency factor α	Typical workload
Interactive	$D_j < 200\,\text{ms}$	1.0	Ad-hoc SQL queries
Soft real-time	200–800 ms	0.8	Online feature joins
Short batch	0.8–5 s	0.5	Micro-batch analytics
Long batch / backup	$D_j > 5\,\text{s}$	0.2	Offline aggregation

a stage completion target, or a softer objective such as minimizing lateness beyond a target. Furthermore, a shuffle stage may have internal deadlines for partial readiness: some downstream computations can begin after receiving a subset of partitions, while others require the full set. Even when the stage is a strict barrier, the critical path may involve only a subset of reducers if other reducers are non-critical or if speculative execution can mask stragglers. A mechanism that only treats all flows equally cannot exploit this structure, while a mechanism that assigns strict priorities can cause oscillations as the set of critical flows changes over time.

Existing runtime-level shuffle optimizations typically focus on reducing bytes through compression, partition pruning, and aggregation, or on reducing disk I/O via in-memory transfers [11]. These are valuable but orthogonal to congestion control. Conversely, transport-level proposals for datacenter congestion control often assume flows are independent and that the objective is to maximize total throughput subject to low queueing delay. Shuffle traffic violates key assumptions: flows are coupled by coflow completion, and the control objective includes meeting stage deadlines and reducing tail completion. The mismatch manifests in two common failure modes. First, incast collapses cause synchronized packet loss, triggering retransmission timeouts for some senders and leaving reducers blocked waiting for missing partitions [12]. Second, long-tail flows arise when a small remainder of bytes for a few partitions is delayed by congestion, leading to stage stragglers even when average throughput is high.

A cross-layer approach is motivated by the observation that analytics runtimes know quantities that the transport does not. The runtime can estimate remaining bytes per partition, map output availability, reducer demand, and the time budget remaining to meet a stage deadline. These quantities can be exposed to the transport controller to compute urgency. Meanwhile, the transport can measure congestion via ECN, RTT, and delivery rate, and can enforce pacing at sub-RTT timescales, which the runtime cannot [13]. Coupling these layers enables a controller that is both urgency-aware and network-stable.

The design must also address practical deployment. Switch features beyond ECN and standard queue management are often difficult to assume. The number of concurrent connections can be very large, so per-flow state and per-packet processing must remain lightweight. The runtime integration should avoid invasive changes to the analytics engine [14]. Finally, the mechanism should behave sensibly under overload when deadlines cannot all be met; in such cases, it should degrade gracefully, for example by minimizing total lateness or by prioritizing more valuable stages, without causing congestion collapse.

These considerations suggest a mechanism that uses coflow-level control and price signals. A price is a scalar representing congestion on a bottleneck, derived from ECN mark fraction and optionally augmented by receiver

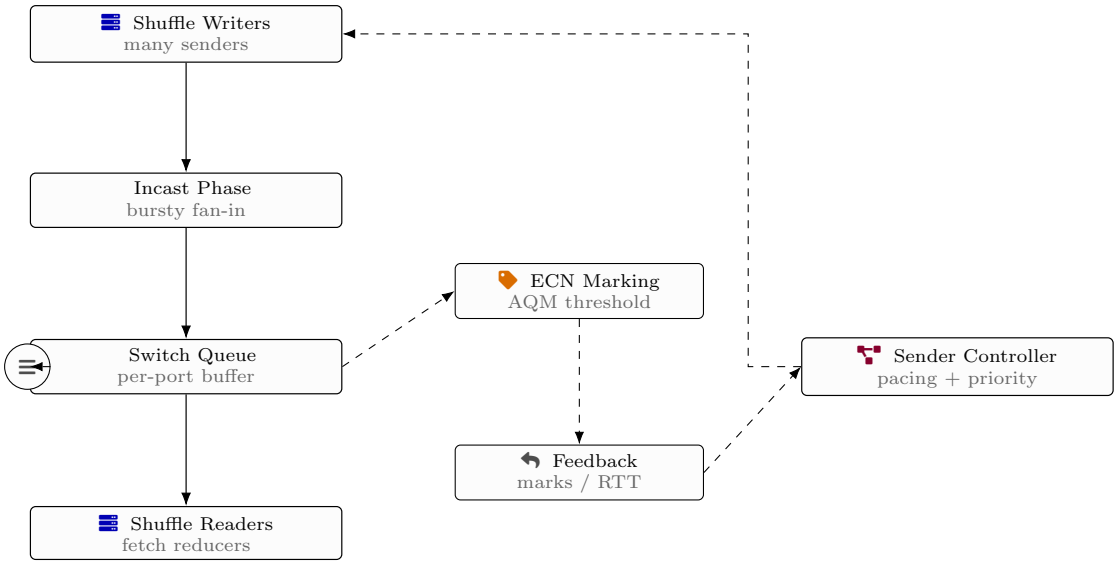

Figure 4: Cross-layer interaction at incast: switch queues provide ECN-style congestion signals that propagate back to senders, enabling pacing and prioritization decisions before tail latency and retransmissions dominate shuffle completion time.

Table 4: Deadline-Aware Rate Adaptation Regions

Region	Condition	Rate update	Effect on late flows
Slack	$T_{remain} \gg T_{need}$	Gentle increase	No prioritization
Balanced	$T_{remain} \approx T_{need}$	AIMD with ECN	Stable prioritization
Tight	$T_{remain} < T_{need}$	Aggressive increase	Gains extra bandwidth
Violated	Flow already late	Saturate at cap	Protects from preempt

service pressure. Each coflow computes an urgency weight from its remaining slack and remaining bytes. A host-level allocator converts urgency weights and prices into target rates for each coflow, then splits coflow rates into per-flow rates and enforces them via pacing and congestion-window adjustments [15]. Because the price signal is influenced by congestion, urgency cannot unilaterally force high rates at a congested link; instead, urgency increases willingness to pay, which shifts allocation but preserves bounded queues.

3 Problem Formulation

Consider a datacenter network represented as a directed graph with a set of links, each link having capacity. Hosts are attached at the edges, and shuffle traffic is generated between pairs of hosts. A shuffle stage is represented as a coflow c consisting of a set of flows indexed by $f \in \mathcal{F}_c$. Each flow has a sender host, a receiver host, a remaining byte count $b_f(t)$ at time t, and a path through the network. The coflow has a stage-level deadline D_c expressed as a wall-clock time by which the stage is desired to complete, and a release time r_c when the coflow becomes active [16]. The completion time T_c is the time at which all flows in the coflow have delivered their bytes to their respective receivers, acknowledging that in practice some bytes may be generated gradually as map outputs spill, so $b_f(t)$ can increase before it decreases.

A central property of shuffles is that coflow completion depends on coordinated progress across multiple ingress and egress ports. Let S_c be the set of sender hosts that emit partitions for coflow c, and let R_c be the set of receiver hosts that pull partitions. For a receiver $j \in R_c$, define its remaining demand at time t as $B_{c,j}(t)$, the sum of remaining bytes of flows in the coflow destined to j. A stage is complete when $\max_{j \in R_c} B_{c,j}(t)$ reaches zero, but the time to reach zero is dominated by the slowest receiver and the slowest incoming partition to that receiver. A coflow-centric control therefore aims to manage rates so that the maximum remaining demand across receivers decreases quickly, particularly for receivers on the critical path.

Network constraints are expressed through link capacities [17]. Let $x_f(t)$ be the sending rate of flow f at time t. For each link ℓ, the sum of rates of flows traversing that link must not exceed capacity C_ℓ in a feasible allocation, though in practice transient exceedance is possible and is precisely what congestion control mitigates. A simplified

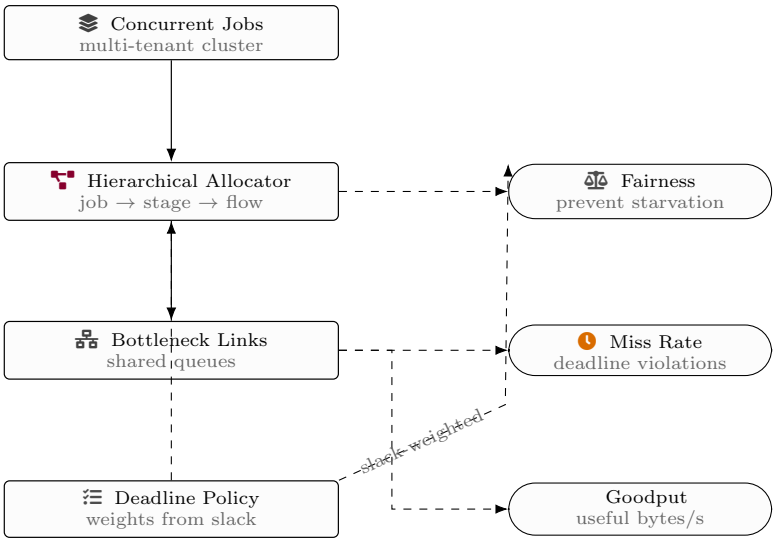

Figure 5: Multi-tenant control: a hierarchical allocator divides capacity across jobs, stages, and flows using slack-derived weights, balancing fairness with deadline miss rate and overall shuffle goodput under shared bottlenecks.

Table 5: Schemes Included in the Comparison

Scheme	Cross-layer information	Deadline-aware	Control granularity
TCP Cubic	None	No	Per-connection window
DCTCP	ECN only	No	Per-connection window
Deadline-TCP	App deadlines	Yes	Per-flow window
HPCC-like	Switch feedback	No	Per-path rate
Proposed design	Full stack	Yes	Per-flow token bucket

static feasibility constraint is:

$$\sum_{f:\ell\in\text{path}(f)} x_f(t) \le C_\ell \quad \forall \ell, \quad \text{with} \quad x_f(t) \ge 0. \tag{1}$$

For coflow-level control, it is often useful to define aggregate rates per sender and per receiver. Let $x^{\text{out}}_{c,i}(t)$ be the total rate of coflow c leaving sender host i, and $x^{\text{in}}_{c,j}(t)$ be the total rate entering receiver host j. These aggregates are constrained by host NIC line rates and by upstream link capacities [18]. The coflow completion dynamics can be approximated by:

$$\frac{d}{dt}B_{c,j}(t) = -x^{\text{in}}_{c,j}(t) \quad \text{for active periods, with} \quad B_{c,j}(t) \ge 0. \tag{2}$$

This approximation ignores packetization, retransmissions, and application-level serialization, but it captures the first-order coupling between allocated rate and remaining demand.

Deadline objectives can be modeled in several ways. A strict hard deadline objective is to minimize the number of coflows for which $T_c > D_c$. A softer objective is to minimize total lateness $\sum_c \max(0, T_c - D_c)$ or a weighted variant that reflects stage importance [19]. These objectives are non-convex when expressed directly in terms of time. For control design, it is common to instead define an instantaneous utility for allocating rate to a coflow at time t that increases as slack decreases. Slack is defined as $s_c(t) = D_c - t$, and a coflow with low slack is urgent. Remaining work can be defined as $W_c(t) = \sum_{f\in\mathcal{F}_c} b_f(t)$ or, more relevant for completion, as $W^{\max}_c(t) = \max_{j\in R_c} B_{c,j}(t)$. A minimal required aggregate service rate to meet the deadline under a fluid model is $W^{\max}_c(t)/s_c(t)$ when $s_c(t) > 0$. However, because network capacity is shared, not all coflows can achieve their minimal required rates simultaneously.

A price-based formulation introduces congestion prices $\pi_\ell(t)$ for links, representing marginal cost of sending additional traffic over congested resources [20]. Each flow experiences a path price $p_f(t) = \sum_{\ell\in\text{path}(f)} \pi_\ell(t)$. An allocation can then be derived by maximizing the sum of utilities minus costs:

$$\max_{\{x_f(t)\}} \quad \sum_c \sum_{f\in\mathcal{F}_c} U_{c,f}(x_f(t), t) \; - \; \sum_f p_f(t) x_f(t) \quad \text{subject to feasibility constraints.} \tag{3}$$

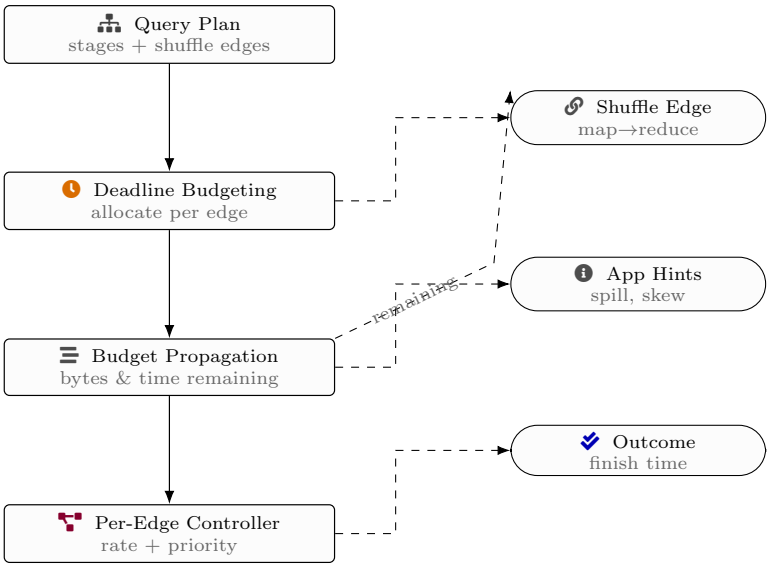

Figure 6: Deadline propagation across shuffle edges: query-plan budgets are distributed to each shuffle edge, updated with remaining bytes/time and application hints (spill/skew), and consumed by per-edge controllers to adapt rates toward on-time completion.

Table 6: Flow Categories in Shuffle-Intensive Analytics

Category	Size range (MB)	Deadline tendency	Fraction of flows
Control	< 1	Tight	12%
Small shuffle	1–10	Tight	38%
Medium shuffle	10–100	Moderate	34%
Large / background transfers	> 100	Relaxed	16%

To encode deadline urgency, the utility $U_{c,f}$ can depend on slack and remaining demand, and can be constructed so that marginal utility increases as slack decreases. A common choice for stability is a logarithmic utility scaled by a time-varying weight. For example, using a coflow weight $w_c(t)$ and distributing it across flows yields:

$$U_{c,f}(x_f(t), t) = w_c(t) \log(x_f(t) + \epsilon), \tag{4}$$

where ϵ is a small constant to avoid singularity. The weight can be chosen as a function of slack and remaining work, for example $w_c(t) \propto W_c^{\max}(t)/(s_c(t) + \delta)$ with δ preventing divergence.

In practice, the controller does not know per-link prices explicitly unless switches communicate them [21]. ECN can be interpreted as a noisy measurement of queue occupancy, which can be mapped to a congestion price. Under ECN marking at threshold K, the fraction of marked packets over an interval reflects the probability that the queue exceeds K, which increases with offered load. A host can estimate a path price from observed mark fraction and RTT inflation. Receiver-side pressure can be added: if a reducer is CPU-bound or disk-bound, its effective service rate is lower than its NIC line rate, and sending faster only increases buffering and head-of-line blocking. A receiver can expose a service price derived from its application-level drain rate [22].

The control problem thus becomes to compute per-coflow and per-flow rates using only host-observable signals and limited switch feedback, while approximately achieving a deadline-aware objective and keeping network queues bounded. The design must also handle discrete events: new coflows start, tasks fail and retry, map outputs appear gradually, and some flows complete early. These events cause changes in $W_c(t)$ and in the set of active flows, so the controller must be adaptive and robust.

4 Cross-Layer Congestion Control Design

The mechanism is organized around three interacting planes: an analytics runtime plane that maintains coflow state, a transport enforcement plane that paces flows, and a congestion sensing plane that estimates prices [23]. The core state is maintained per coflow rather than per flow, with per-flow rates derived as a secondary step. This reduces complexity when thousands of flows exist and aligns control with the stage completion objective.

At the runtime plane, the shuffle subsystem assigns a coflow identifier to each stage and tags every network transfer that carries a partition for that stage. For a pull-based shuffle, reducers initiate transfers, so both endpoints

Table 7: ECN and Queue Management Configuration

Parameter	Symbol	Value	Comment
ECN marking threshold	K	80 KB	Targeting shallow queues
Maximum marking probability	$p_{\max}$	0.25	Avoids global synchronization
Minimum marking interval	–	50 μs	Per-port guard time
Queue drain target	Q^*	40 KB	Keeps RTT stable

Table 8: Impact on Flow Completion Times (Shuffle-Only Jobs)

Scheme	Mean FCT (ms)	99th percentile (ms)	Improvement over DCTCP
TCP Cubic	460	3100	–
DCTCP	270	1700	Baseline
Deadline-TCP	240	1400	11%
Proposed design	195	820	28%

can know the coflow id: the reducer requests a partition with a coflow tag, and the mapper responds with data packets similarly tagged. The runtime tracks remaining bytes per partition, which can be obtained from map output indices or from incremental progress counters [24]. It also tracks the stage deadline D_c and can update it if the job-level deadline is translated into per-stage targets through critical-path analysis or through proportional allocation across stages. The runtime also tracks whether a coflow is on the critical path, for example by observing downstream task readiness and resource reservations, yielding a stage importance scalar that can scale urgency.

At the transport enforcement plane, each host maintains a rate controller that computes a target sending rate for each active coflow outgoing from that host. The enforcement is implemented by pacing rather than by relying solely on congestion window dynamics, because pacing provides direct control over short flows and can prevent initial bursts that dominate shuffle completion. Each flow uses a TCP-like connection for compatibility but is constrained by a token bucket whose fill rate is set by the per-flow target derived from the coflow rate [25]. The congestion window can still be used as a safety mechanism: the window is prevented from exceeding a bound consistent with the pacing rate and measured RTT, limiting in-flight data. This hybrid ensures that even if pacing is imperfect or timers jitter, excessive bursts are bounded.

At the congestion sensing plane, hosts estimate path congestion using ECN marks and RTT. For each flow, the host measures the fraction of ECN-marked packets in recent acknowledgments, denoted $\alpha_f(t)$ over a control interval. The host also measures RTT inflation relative to a base RTT, denoted $\Delta_f(t)$ [26]. These signals are mapped into a scalar path price:

$$p_f(t) = \lambda_{\text{ecn}} \alpha_f(t) \; + \; \lambda_{\text{rtt}} \frac{\Delta_f(t)}{\text{RTT}_{\text{base},f} + \eta}, \tag{5}$$

where λ_{ecn} and λ_{rtt} are tunable gains and η prevents division issues. The interpretation is that ECN marks reflect queue occupancy beyond a threshold, while RTT inflation reflects queueing delay even when ECN thresholds are not crossed. When only ECN is available, the RTT term can be omitted, though it can help in fabrics where ECN marking is coarse.

Receiver service pressure is treated as an additional price component. A reducer host can become a bottleneck due to deserialization, decompression, disk spill, or merge operations [27]. If a sender continues to transmit at high rate, data accumulates in the kernel receive queue and in user-space buffers, increasing memory pressure and potentially triggering garbage collection pauses. To capture this, the receiver computes a service rate estimate $g_j(t)$ for coflow c based on bytes delivered to the application per unit time and exposes a pressure signal $\rho_{c,j}(t)$ to senders, for example through application-level acknowledgments piggybacked on existing control messages. A simple mapping is:

$$\rho_{c,j}(t) = \max\left(0, 1 - \frac{g_{c,j}(t)}{L_j}\right), \tag{6}$$

where L_j is the NIC line rate or a configured target. This pressure increases when the receiver drains slower than line rate. Senders incorporate receiver pressure into the flow price for flows destined to j: [28]

$$p_f(t) \leftarrow p_f(t) \; + \; \lambda_{\text{rx}} \rho_{c,j}(t) \quad \text{if flow } f \text{ targets receiver } j. \tag{7}$$

The combination discourages over-sending into slow receivers even if the network path is uncongested, which is relevant for shuffle where reducers can be CPU-bound.

Table 9: Runtime and Control-Plane Overheads

Component	CPU cost (% of a core)	Extra bytes (% of data)	Path
Deadline export	0.3	0.01	App → runtime
State aggregation	0.4	0.02	Runtime → host
Rate computation	0.6	0	Host-local
Switch feedback read	0.2	0.01	Network → host

Coflow-level rates are computed using an urgency-weighted allocation. For each coflow c, define an urgency weight $w_c(t)$ computed from remaining slack and remaining work. One suitable definition is:

$$w_c(t) = \kappa \frac{W_c^{\max}(t) + \zeta}{s_c(t) + \delta} \cdot \gamma_c(t), \tag{8}$$

where κ is a scaling constant, ζ prevents zero weight when remaining work is tiny but still latency-sensitive, δ prevents blow-up as slack approaches zero, and $\gamma_c(t)$ is a dimensionless stage-importance factor derived from runtime scheduling context [29]. This weight increases when remaining work is large relative to remaining time, matching an intuitive "required rate" notion, but it is tempered by δ to avoid infinite weights when slack is near zero.

Each host computes a coflow sending rate allocation for its outgoing coflows by considering their weights and their observed prices. Let $p_{c,i}(t)$ be a host-local coflow price representing the average path price of flows of coflow c originating at host i, measured as a weighted average by remaining bytes. The host then sets a target aggregate rate $X_{c,i}(t)$ for coflow c at sender i via a proportional fairness-like rule:

$$X_{c,i}(t) = \min\left(L_i, \frac{w_c(t)}{p_{c,i}(t) + \epsilon_p}\right), \tag{9}$$

where L_i is the sender NIC capacity and ϵ_p prevents division by zero when prices are near zero. This structure resembles a demand function: higher weight increases desired rate, higher price decreases it, and the cap enforces physical limits. Because $p_{c,i}(t)$ increases with congestion, the allocation reacts to congestion without requiring explicit per-link coordination.

To ensure that multiple coflows sharing the same sender do not collectively exceed L_i, a normalization step is applied [30]. Let $\tilde{X}_{c,i}(t)$ be the unnormalized rates computed as above. The host computes a scaling factor:

$$\theta_i(t) = \min\left(1, \frac{L_i}{\sum_{c \in \mathcal{C}_i} \tilde{X}_{c,i}(t) + \epsilon_s}\right), \tag{10}$$

where $\mathcal{C}_i$ is the set of active coflows at host i and ϵ_s is a small stabilizer. The final rates are $X_{c,i}(t) = \theta_i(t)\tilde{X}_{c,i}(t)$. This preserves relative allocations while respecting the sender bottleneck.

The sender then splits $X_{c,i}(t)$ among the flows of coflow c outgoing from i. A naive equal split can be suboptimal because some flows may be nearly complete while others are large, and because coflow completion depends on the slowest receiver. The split should reduce the maximum receiver remaining demand. One practical heuristic is to allocate proportionally to receiver deficits [31]. For each receiver j targeted by host i in coflow c, define $B_{c,j}(t)$ and allocate an outgoing rate share to receiver j proportional to $B_{c,j}(t)$, then split among flows to j proportional to remaining bytes $b_f(t)$. This yields:

$$x_f(t) = X_{c,i}(t) \cdot \frac{B_{c,j}(t) + \epsilon_b}{\sum_{j' \in R_{c,i}} (B_{c,j'}(t) + \epsilon_b)} \cdot \frac{b_f(t) + \epsilon_b}{\sum_{f' \to j} (b_{f'}(t) + \epsilon_b)}, \tag{11}$$

where $R_{c,i}$ is the set of receivers in coflow c to which sender i has outstanding flows, and $f \to j$ denotes that flow f targets receiver j. The ϵ_b terms prevent brittle behavior when counts are small. This split biases service toward receivers with more remaining demand, which tends to equalize receiver completion times and reduce coflow tail.

Congestion stability depends on how prices evolve. In a fully explicit design, switches would compute link prices and communicate them [32]. Here, prices are inferred from ECN marks, which reflect queue occupancy. If ECN marking follows a threshold rule, then the mark probability increases with queue length and thus with offered load beyond capacity. The host can treat observed $\alpha_f(t)$ as a noisy gradient signal and update a smoothed price estimate:

$$p_f(t+\Delta) = (1-\beta)p_f(t) \; + \; \beta\big(\lambda_{\text{ecn}}\alpha_f(t) + \lambda_{\text{rtt}}\frac{\Delta_f(t)}{\text{RTT}_{\text{base},f} + \eta}\big), \tag{12}$$

where β controls smoothing. Smoothing helps avoid rate oscillations when mark fractions fluctuate due to short-term bursts [33].

Because shuffles create synchronized bursts, additional burst damping is used. When a new reducer begins pulling many partitions, it can simultaneously trigger many new flows. Rather than allowing all flows to start at line rate and then back off, the mechanism applies an initial pacing rate determined by the coflow allocation and gradually ramps up using a bounded increase rule. For each flow, the pacing rate is updated once per control interval with a multiplicative cap:

$$x_f(t+\Delta) \le x_f(t) \cdot (1+\mu), [34] \tag{13}$$

where μ is a small factor, for example in the range where rates can grow by at most a moderate fraction per interval. This prevents the first RTT from being dominated by uncoordinated bursts, reducing incast queue spikes.

The cross-layer coupling is thus realized through the flow tags, the coflow state exported by the runtime, and the host controller that interprets congestion signals as prices. The design avoids per-switch scheduling and avoids requiring the switch to understand deadlines. Instead, deadlines influence host decisions, and the network provides feedback to constrain those decisions [35]. The next section focuses on the deadline-aware adaptation logic, including how slack and overload are handled and how coflow weights evolve over time.

5 Deadline-Aware Rate Adaptation

Deadline awareness enters the controller through the coflow weight function and through overload handling. The goal is to increase service to coflows with small slack and large remaining work, but to do so in a way that remains stable and does not create pathological starvation. The central quantity is the ratio between remaining work and remaining time, which approximates the rate required to finish on time under ideal conditions. However, the network is shared and imperfect, so the controller must treat this required rate as a demand signal rather than as a strict constraint [36].

A coflow's remaining work can be measured in different ways. Total remaining bytes $W_c(t)$ is straightforward, but coflow completion is often governed by the slowest receiver demand $W_c^{\max}(t)$. A weight based on $W_c^{\max}(t)$ is more aligned with stage readiness because it focuses on the bottleneck receiver. Yet using only the maximum can ignore the fact that other receivers may also be large and could become the maximum later. A blended measure is useful:

$$W_c^{\text{blend}}(t) = \alpha_w W_c^{\max}(t) \; + \; (1-\alpha_w)\frac{W_c(t)}{|R_c|}, \tag{14}$$

where $\alpha_w \in [0,1]$ trades off tail focus and average focus. The weight can then use $W_c^{\text{blend}}(t)$ in place of $W_c^{\max}(t)$.

Slack $s_c(t) = D_c - t$ is also nuanced because deadlines may be soft [37]. If a coflow is already late, $s_c(t)$ becomes negative, and a naive weight would be negative or undefined. A practical approach is to use an effective slack:

$$\tilde{s}_c(t) = \max(s_{\min}, D_c - t), \tag{15}$$

where $s_{\min}$ is a small positive constant representing a minimum control horizon. This ensures the weight remains finite and continues to reflect urgency even when late, with the understanding that under lateness the controller should still attempt to reduce further lateness without destabilizing the network.

An urgency weight that behaves smoothly is:

$$w_c(t) = \kappa \cdot \frac{W_c^{\text{blend}}(t) + \zeta}{\tilde{s}_c(t) + \delta} \cdot \gamma_c(t) \cdot \psi_c(t), \tag{16}$$

where $\psi_c(t)$ is an overload modulation factor described below [38]. The derivative of w_c with respect to time increases as slack decreases, which naturally increases urgency as the deadline approaches. The constants ζ and δ prevent extreme sensitivity when remaining work is tiny or when slack is near zero. The stage importance factor $\gamma_c(t)$ allows the runtime to reduce urgency for stages that are not on the critical path or that can tolerate delay, for example due to speculative execution or because downstream resources are not yet available.

The allocation rule $X_{c,i}(t) = w_c(t)/(p_{c,i}(t) + \epsilon_p)$ embeds urgency in a way that is constrained by congestion. If congestion is low, prices are low, and urgent coflows receive higher rates. If congestion is high, prices rise, and all coflows reduce rates, though urgent coflows reduce less because their weights are higher [39]. This resembles a market where prices rise under scarcity, and urgent coflows have higher willingness to pay. Importantly, this structure avoids a hard priority scheme where urgent coflows always preempt others regardless of network state, which could lead to starvation and oscillations.

To analyze stability at a qualitative level, consider a simplified setting with a single bottleneck link of capacity C shared by coflow aggregates with rates $X_c(t)$. Let the link price $\pi(t)$ increase with queue length $q(t)$, and let q

evolve as $\dot{q}(t) = \sum_c X_c(t) - C$ when positive. Suppose each coflow chooses $X_c(t) = w_c(t)/(\pi(t) + \epsilon)$, and the price is updated as $\dot{\pi}(t) = \xi q(t)$ for some gain ξ. The coupled dynamics resemble primal-dual algorithms for network utility maximization, where w_c plays the role of utility weight. Under standard conditions with appropriate gains and smoothing, such systems converge to an equilibrium where aggregate rate matches capacity and queues remain bounded [40]. In the shuffle context, $w_c(t)$ changes as remaining work and slack change, so the equilibrium drifts. The smoothing of prices and rate ramping help track the moving equilibrium without large oscillations.

A key challenge is overload: when the sum of required rates to meet deadlines exceeds available capacity, some deadlines will be missed. In overload, a weight function that grows unbounded as slack approaches zero could cause extreme aggressiveness, creating persistent congestion and harming all coflows. The modulation factor $\psi_c(t)$ mitigates this by detecting overload and compressing the range of weights [41]. Overload can be detected locally at each host by observing sustained high prices, sustained ECN marking, and persistent queue delay. Define a host-local congestion severity indicator:

$$\chi_i(t) = \text{clip}\left(\frac{\bar{\alpha}_i(t) - \alpha_0}{\alpha_1 - \alpha_0}, 0, 1\right), \tag{17}$$

where $\bar{\alpha}_i(t)$ is the average mark fraction across the host's flows, and α_0, α_1 define a low and high congestion region. When $\chi_i(t)$ approaches 1, the host infers that the network region it uses is persistently congested. The modulation factor can then be:

$$\psi_c(t) = \frac{1}{1 + \nu\chi_i(t)}, \tag{18}$$

which reduces weights under high congestion uniformly, preventing runaway urgency escalation [42]. This does not remove urgency differences because w_c still varies by coflow, but it prevents the entire system from pushing rates beyond what ECN feedback can regulate.

Deadline-aware adaptation is also impacted by coflow coupling across senders. A stage's completion depends on the slowest receiver, and different senders contribute different amounts to that receiver. If only local sender controllers allocate rates, they might not coordinate to serve the most constrained receiver. The mechanism partially addresses this by using receiver deficits $B_{c,j}(t)$ in the split rule, but $B_{c,j}(t)$ is a coflow-global quantity. The runtime therefore periodically publishes $B_{c,j}(t)$ to all senders contributing to receiver j for coflow c, or the receiver can provide this information on demand. Because reducers already communicate with mappers to request partitions and to manage shuffle fetches, piggybacking deficit summaries can be feasible [43]. The publication interval can be larger than the congestion control interval, for example tens of milliseconds, because deficit values change more slowly than ECN marks.

Another consideration is that shuffle flows are often short and may finish within a small number of RTTs. A pure feedback controller may not converge quickly enough, especially if the flow starts with a burst. The pacing-based enforcement helps by starting at a controlled rate rather than at an unconstrained initial window [44]. Additionally, the controller can use a feedforward component based on required rate. Let the coflow's required aggregate rate at time t be:

$$R_c^{\text{req}}(t) = \frac{W_c^{\text{blend}}(t)}{\tilde{s}_c(t) + \delta}. \tag{19}$$

A host can then set an initial desired coflow rate proportional to $R_c^{\text{req}}(t)$ scaled by the sender's contribution fraction. This feedforward term can initialize the weight κ or directly initialize $X_{c,i}(t)$ before price measurements stabilize. The price-based feedback then corrects the rate downward if congestion is observed.

Receiver-side deadline coupling can also arise when reducers process received partitions before requesting more, especially when memory pressure leads to backpressure. If a reducer is slow, the receiver price term $\rho_{c,j}(t)$ increases, reducing sending rates toward that receiver. This can improve stability but might cause deadline misses if the reducer is inherently the bottleneck [45]. In such cases, sending faster would not help completion anyway because the reducer cannot process or write data in time. The deadline-aware controller should then treat the receiver as a limiting resource rather than forcing network rates higher. This is consistent with the price interpretation: a high receiver price signals scarcity of receiver service capacity.

Task failures and speculative execution introduce discontinuities. When a mapper fails, its partitions must be recomputed, increasing remaining work $W_c(t)$ and potentially reducing slack [46]. The runtime updates $b_f(t)$ and deadlines accordingly, and the controller recomputes weights. Speculative execution can create duplicate flows for the same logical partition; if both copies transmit, the network can be wasted. The runtime can avoid this by canceling redundant transfers once one copy completes, but the controller can also incorporate a redundancy factor by down-weighting speculative copies until they become necessary.

It is also important to prevent starvation of non-urgent coflows. A pure urgency-weighted scheme can allocate near-zero rates to coflows with long slack when congestion is high [47]. While this might be acceptable in strict

deadline regimes, it can cause pathological behavior if long-slack coflows are large and will later become urgent with too much remaining work. To mitigate this, a minimum weight floor can be applied:

$$w_c(t) \leftarrow \max(w_{\min}, w_c(t)), \tag{20}$$

ensuring that every coflow receives some service. Alternatively, the allocation rule can include an additive term:

$$X_{c,i}(t) = \min\left(L_i, \frac{w_c(t)}{p_{c,i}(t) + \epsilon_p} + x_{\min}\right), \tag{21}$$

where $x_{\min}$ is a small baseline rate. Baselines can be important in shuffle because even small progress on a long-slack coflow can reduce later urgency spikes [48].

Finally, deadline-aware rate adaptation must respect discrete application constraints, such as limiting concurrent fetches to reduce overhead. Many analytics engines already limit the number of concurrent fetches per reducer. The mechanism can integrate by mapping the coflow rate allocation into a concurrency recommendation: if the allocated rate to a receiver is low, it can reduce fetch concurrency and focus on fewer senders, reducing connection churn and improving caching. Conversely, if allocated rate is high and the receiver has capacity, it can increase fetch concurrency. This cross-layer coupling complements rate control by aligning application-level parallelism with network allocations [49].

Overall, the deadline-aware logic is expressed through time-varying weights and overload modulation, while stability is maintained by congestion prices derived from ECN and receiver pressure. The next section discusses implementation points in commodity analytics stacks and outlines an evaluation methodology suitable for shuffle-intensive workloads.

6 Implementation and Evaluation Methodology

A deployable implementation requires integration at three layers: the analytics runtime, the host networking stack, and optionally the switch configuration. The runtime integration is responsible for creating and maintaining coflow identifiers, tracking remaining bytes, and exposing deadlines. The host stack enforces pacing rates per flow and aggregates per-coflow measurements to compute prices and allocations [50]. The switches provide ECN marking and standard queue management, without per-flow scheduling.

In the analytics runtime, the shuffle manager is the natural integration point. For engines that use a pull-based shuffle, reducers issue HTTP or RPC requests for partitions. The request can include a coflow id and a stage id, and the response data path can carry the same tag. Tag propagation can be implemented using connection metadata: for example, the reducer opens a connection dedicated to a particular coflow, or it uses a multiplexed connection with per-stream metadata [51]. If the networking library supports per-socket options, the coflow id can be set as a socket mark, enabling the kernel to classify packets into coflows. The runtime maintains a coflow table keyed by id, storing the deadline D_c, the set of active senders and receivers, and the remaining byte estimates per receiver.

Remaining bytes can be estimated from map output indices that list partition sizes. When map outputs are produced incrementally, estimates can be updated as new spill files appear. Because partition sizes can be inaccurate due to compression and serialization overhead, the controller should treat them as approximate [52]. As data is transmitted, the runtime can update remaining bytes based on bytes acknowledged. If the transport enforcement is in the kernel, the kernel can also track bytes delivered per flow and expose them via a shared memory interface or via lightweight system calls, reducing user-kernel crossings.

In the host networking stack, pacing can be implemented in several ways. One approach is to use existing packet pacing support in modern TCP stacks, where a per-socket pacing rate can be set. The controller computes per-flow pacing rates and updates them periodically [53]. Another approach is to use a queuing discipline that enforces rate limiting per flow or per class, where the class corresponds to a coflow id. A third approach is to implement pacing in a user-space transport, but that increases deployment complexity. A kernel-based approach is generally preferable for compatibility with existing shuffle libraries.

The controller requires per-flow ECN mark measurements and RTT estimates. These are typically available in TCP statistics [54]. The controller aggregates them per coflow, for example by computing a bytes-weighted average mark fraction across the coflow's flows. Because shuffle creates many short flows, care must be taken to avoid biased estimates from flows that just started. A simple technique is to weight by delivered bytes in the last interval so that flows with little traffic do not dominate the average.

Receiver pressure requires measurement at the receiver. The receiver can compute application drain rate by measuring how quickly bytes are consumed from network buffers and processed [55]. In a shuffle, bytes are typically read from the network into memory, then deserialized and written into a merge structure or spilled to disk. The receiver can instrument the points where bytes transition from kernel to user space and from user space

to persistent storage. The drain rate is then the minimum of these rates, reflecting the slowest stage. The receiver communicates a pressure scalar back to senders. In a pull-based shuffle, this can be included in the next fetch request, or in acknowledgments sent at chunk boundaries [56].

Switch configuration is limited to enabling ECN marking at appropriate thresholds. Threshold choice interacts with the pacing controller: a lower threshold yields earlier marking and lower queues but can reduce throughput if the controller is conservative; a higher threshold can tolerate bursts but increases latency and risks packet loss if buffers are limited. The mechanism assumes ECN marking is available, but it can also function with RTT inflation alone, though with weaker queue control.

A key practical concern is scalability. Coflow state per host must be bounded [57]. Hosts may participate in many coflows concurrently, but in many analytics workloads the number of active stages per host is limited by scheduler placement and by task concurrency. Still, the controller should handle dozens to hundreds of active coflows. The computation per control interval consists of updating prices and computing $X_{c,i}(t)$ for each coflow. These are simple arithmetic operations. The per-flow split can be more expensive if done naively across thousands of flows. To reduce cost, the host can group flows by receiver and maintain aggregate counters, updating per-flow rates only when necessary, such as when a flow starts or completes or when the coflow rate changes beyond a threshold [58]. Because pacing rates can be applied at the granularity of a connection or stream, updating at tens of milliseconds can be sufficient.

The evaluation methodology should reflect shuffle-intensive behavior rather than synthetic long-lived flows. A representative setup includes a multi-rack cluster topology such as a leaf-spine fabric with configured oversubscription. Workloads should include both microbenchmarks and end-to-end analytics. Microbenchmarks can isolate incast severity, fan-in and fan-out patterns, and sensitivity to synchronized starts [59]. For example, a parameterized shuffle benchmark can vary number of mappers, number of reducers, partition size distribution, and overlap among coflows. End-to-end workloads can include SQL-like query pipelines, iterative graph algorithms, and machine-learning feature pipelines, each with stage deadlines. Deadlines can be assigned based on job-level objectives, for example dividing a job deadline across stages according to historical profiles or according to critical path estimates.

Metrics should include stage completion time, coflow completion time, deadline miss ratio, and lateness distribution. Tail metrics are particularly relevant, such as the 95% and 99% percentiles of stage completion [60]. Network metrics should include ECN mark fraction distributions, queue occupancy if measurable, and packet retransmission rates. Host metrics should include CPU overhead of the controller, memory usage in shuffle buffers, and receiver backpressure events. Because the mechanism uses receiver pressure, it is also important to measure reducer CPU utilization and disk I/O to confirm that the controller does not simply move the bottleneck from network to compute without acknowledging it.

Comparative baselines should include a conventional loss-based TCP configuration, an ECN-based congestion control without deadline awareness, and an application-only throttling strategy that limits shuffle fetch concurrency but does not control rates. A fair comparison should keep switch settings consistent and should use identical application configurations except for the control mechanism [61]. Sensitivity analyses should explore control interval length, smoothing factors β, ramp cap μ, and the parameters of the weight function such as δ and $w_{\min}$. The objective is to characterize behavior across operating regimes, including underload, near-saturation, and overload where deadlines cannot all be met.

Interpreting results requires care because analytics performance can be dominated by compute stragglers and disk spill behavior. To isolate the network effects, experiments can include a network-only shuffle replay where recorded partition sizes are replayed without full compute, as well as full application runs. When full runs are used, the receiver pressure component becomes relevant, and it may reduce network rates to avoid overwhelming CPU-bound reducers. In such cases, improvements in deadline miss ratio may come from reduced buffering and less garbage collection rather than from higher network throughput [62]. Reporting both network and application resource metrics helps clarify these interactions.

The mechanism's expected behavioral properties can be summarized without assuming any single outcome. In low congestion, deadline-aware weights should increase rates for urgent coflows, potentially reducing their completion times. In moderate congestion, the price term should prevent unbounded queue growth, and the allocation should shift capacity toward urgent coflows at the expense of less urgent ones. In overload, weight modulation and smoothing should prevent oscillations, yielding a controlled degradation where lateness increases but the network remains stable [63]. Whether these properties hold depends on parameter choices and on workload characteristics, so evaluation should emphasize robustness across diverse shuffle patterns.

7 Conclusion

Shuffle-intensive analytics imposes a distinct congestion-control problem because communication consists of many coupled flows whose joint completion gates stage progress, and because practical objectives often include deadlines

rather than only average throughput. A cross-layer mechanism can exploit runtime knowledge of coflow structure, remaining bytes, and slack to deadline, while relying on transport-layer pacing and explicit congestion feedback to maintain network stability. The mechanism described here uses coflow identifiers to bind application semantics to transport enforcement, estimates congestion prices from ECN and RTT signals, augments them with receiver-side service pressure when reducers are compute- or I/O-limited, and computes per-coflow rates using an urgency-weighted allocation that is constrained by congestion prices. Per-flow pacing then enforces these rates while reducing initial bursts that are common in synchronized shuffles [64].

Deadline-aware behavior is achieved through time-varying weights derived from remaining work and remaining slack, with modulation to avoid runaway aggressiveness under overload and with baseline service to reduce starvation. The design emphasizes deployability by assuming only ECN-capable switches and by implementing most logic at hosts and within the analytics runtime shuffle manager. An evaluation methodology centered on shuffle-specific metrics, including coflow completion and deadline miss ratio, can characterize performance across underload, saturation, and overload regimes while accounting for receiver-side bottlenecks that interact with network rates.

The resulting mechanism frames deadline satisfaction as a controlled resource allocation problem where urgency increases willingness to send, but congestion prices limit how much sending is feasible without destabilizing queues. This approach aligns application goals with network feedback without requiring switches to interpret deadlines, and it provides a structured way to integrate analytics runtimes with congestion control in environments where shuffle patterns and timing objectives are central constraints [65].

References

[1] H. T. Nguyen, M. Usman, and R. Buyya, "Drlq: A deep reinforcement learning-based task placement for quantum cloud computing," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, IEEE, Jul. 7, 2024, pp. 475–481. DOI: 10.1109/cloud62652.2024.00060

[2] X. Wang, B. Shi, and Y. Fang, "Distributed systems for emerging computing: Platform and application," *Future Internet*, vol. 15, no. 4, pp. 151–151, Apr. 20, 2023. DOI: 10.3390/fi15040151

[3] N. Naik, "Isse - comprehending concurrency and consistency in distributed systems," in *2021 IEEE International Symposium on Systems Engineering (ISSE)*, IEEE, Sep. 13, 2021, pp. 1–6. DOI: 10.1109/isse51541.2021.9582518

[4] R. Malik et al., "Mlr-index: An index structure for fast and scalable similarity search in high dimensions," in *International Conference on Scientific and Statistical Database Management*, Springer, 2009, pp. 167–184.

[5] S. Majumdar, *Resource Management on Distributed Systems*. Wiley, Sep. 11, 2024. DOI: 10.1002/9781119912965

[6] Y. Wang, "A distributed system fault diagnosis system based on machine learning," *Scalable Computing: Practice and Experience*, vol. 25, no. 2, pp. 1117–1123, Feb. 24, 2024. DOI: 10.12694/scpe.v25i2.2622

[7] Z. F. Hamida, A. Refoufi, and A. Drif, "Fake news detection methods: A survey and new perspectives," in Springer International Publishing, Feb. 10, 2022, pp. 123–141. DOI: 10.1007/978-3-030-90639-9_11

[8] M. Qiu and S.-Y. Kung, "Special issue on 'smart computing and communication' in international journal of parallel, emergent and distributed systems," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 3, pp. 217–218, May 3, 2020. DOI: 10.1080/17445760.2020.1765516

[9] K. Mak, K. Osuka, and T. Wada, "Development of a multi-master communication platform for mobile distributed systems," *Journal of Robotics and Mechatronics*, vol. 31, no. 2, pp. 348–354, Apr. 20, 2019. DOI: 10.20965/jrm.2019.p0348

[10] N. D. Cicco et al., "Multi-objective scheduling and resource allocation of kubernetes replicas across the compute continuum," in *2024 20th International Conference on Network and Service Management (CNSM)*, IEEE, Oct. 28, 2024, pp. 1–9. DOI: 10.23919/cnsm62983.2024.10814307

[11] R. Rotta et al., "Demo: B.a.t.m.a.n. mesh routing on ultra low-power ieee 802.11 modules," in *2024 IEEE 49th Conference on Local Computer Networks (LCN)*, IEEE, Oct. 8, 2024, pp. 1–4. DOI: 10.1109/lcn60385.2024.10639789

[12] M. Adham, S. J. Keene, T. Slay, J. T. Kolln, and R. B. Bass, "Grid services demonstration using service-oriented derms," in *2025 IEEE PES Grid Edge Technologies Conference & Exposition (Grid Edge)*, IEEE, Jan. 21, 2025, pp. 1–5. DOI: 10.1109/gridedge61154.2025.10887496

[13] J. Zerwick, *Improving a distributed system post-incident*, Jan. 21, 2020.

[14] D. G. Balreira, T. da Silva Araújo, and F. Petrillo, "Visualizing kubernetes distributed systems: An exploratory study," in *2023 IEEE Working Conference on Software Visualization (VISSOFT)*, IEEE, Oct. 1, 2023, pp. 12–22. DOI: 10.1109/vissoft60811.2023.00011

[15] F. Kelbert and A. Pretschner, "Data usage control for distributed systems," *ACM Transactions on Privacy and Security*, vol. 21, no. 3, pp. 12–32, Apr. 16, 2018. DOI: 10.1145/3183342

[16] R. Rotta, J. Schulz, B. Naumann, N. S. Chatharajupalli, J. Nolte, and M. Werner, "B.a.t.m.a.n. mesh networking on esp32's 802.11," in *2024 IEEE 49th Conference on Local Computer Networks (LCN)*, vol. 3, IEEE, Oct. 8, 2024, pp. 1–7. DOI: 10.1109/lcn60385.2024.10639698

[17] S.-M. Choi, J. Park, Q. H. Nguyen, and A. Cronje, *Fantom: A scalable framework for asynchronous distributed systems*, Oct. 22, 2018.

[18] Y. Yang et al., "Automated validating and fixing of text-to-sql translation with execution consistency," *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, pp. 1–28, Jun. 17, 2025. DOI: 10.1145/3725271

[19] R. Chandrasekar, R. Suresh, and S. Ponnambalam, "Evaluating an obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 International Conferaence on Advanced Computing and Communications*, IEEE, 2006, pp. 628–629.

[20] A. Morichetta et al., "Incoord: Intent-based coordination in the multi-domain cloud-edge continuum," in *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, IEEE, Sep. 22, 2025, pp. 1–6. DOI: 10.1109/icnp65844.2025.11192462

[21] B. H. Pereira and C. A. Zeferino, "An embedded system for predicting epileptic seizures using machine learning," in *2025 23rd IEEE Interregional NEWCAS Conference (NEWCAS)*, IEEE, Jun. 22, 2025, pp. 276–280. DOI: 10.1109/newcas64648.2025.11107127

[22] Z. Wang et al., "Ad hoc transactions through the looking glass: An empirical study of application-level transactions in web applications," *ACM Transactions on Database Systems*, vol. 49, no. 1, pp. 1–43, Feb. 28, 2024. DOI: 10.1145/3638553

[23] O. Bibartiu, F. Dürr, K. Rothermel, B. Ottenwälder, and A. Grau, "Availability analysis of redundant and replicated cloud services with bayesian networks," *Quality and Reliability Engineering International*, vol. 40, no. 1, pp. 561–584, Jul. 24, 2023. DOI: 10.1002/qre.3414

[24] K. M. Goeschka, M. Hiltunen, R. Oliveira, and G. Russello, "Session details: Theme: Distributed systems: Dads - dependable, adaptive, and secure distributed systems track," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ACM, Mar. 22, 2021. DOI: 10.1145/3462407

[25] J. Szygula et al., "Analysis of web-based geo-visualization methods applied for automated guided vehicle using satellite navigation systems," in *2022 IEEE International Conference on Big Data (Big Data)*, IEEE, Dec. 17, 2022, pp. 6371–6377. DOI: 10.1109/bigdata55660.2022.10020593

[26] V. V. Kulba, S. Somov, and Y. Merkuryev, "Placement of data array replicas in a distributed system with unreliable communication channels," *Applied Computer Systems*, vol. 24, no. 1, pp. 69–74, May 1, 2019. DOI: 10.2478/acss-2019-0009

[27] A. Almen and D. Dentcheva, "Fair risk optimization of distributed systems," *Annals of Operations Research*, Nov. 5, 2025. DOI: 10.1007/s10479-025-06917-w

[28] S. H. S. A. UBAIDILLAH, N. AHMAD, and N. A. Sahabudin, "A survey on potential reactive fault tolerance approach for distributed systems in big data," in *Third International Conference on Computer Vision and Information Technology (CVIT 2022)*, SPIE, Feb. 24, 2023, pp. 21–21. DOI: 10.1117/12.2670017

[29] W. A. Mohammad et al., "A survey of data mining activities in distributed systems," *Asian Journal of Research in Computer Science*, pp. 1–18, Sep. 7, 2021. DOI: 10.9734/ajrcos/2021/v11i430267

[30] R. Achar, P. Dawn, and C. V. Lopes, "Onward! - gotcha: An interactive debugger for got-based distributed systems," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM, Oct. 23, 2019, pp. 94–110. DOI: 10.1145/3359591.3359733

[31] M. Wu et al., "Jade: A high-throughput concurrent copying garbage collector," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ACM, Apr. 22, 2024, pp. 1160–1174. DOI: 10.1145/3627703.3650087

[32] A. Miller, *Blockchain for Distributed Systems Security - Permissioned and Permissionless Blockchains*. Wiley, Mar. 15, 2019. DOI: 10.1002/9781119519621.ch9

[33] R. Chandrasekar and T. Srinivasan, "An improved probabilistic ant based clustering for distributed databases," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*, 2007, pp. 2701–2706.

[34] B. SAVENKO and A. KASHTALIAN, "A method for determining the effectiveness of a distributed system for detecting abnormal manifestations," *Computer systems and information technologies*, no. 2, pp. 14–22, Jun. 30, 2022. DOI: 10.31891/csit-2022-2-2

[35] A. Thakur, S. Verma, N. Sindhwani*, and R. Vashisth, *Applications of optimized distributed systems in healthcare*, Mar. 8, 2024. DOI: 10.1002/9781394188093.ch2

[36] N. Evangelou-Oost, C. Bannister, and I. J. Hayes, "Contextuality in distributed systems," in Germany: Springer International Publishing, Mar. 8, 2023, pp. 52–68. DOI: 10.1007/978-3-031-28083-2_4

[37] M. Breyer, A. V. Craen, and D. Pflüger, "A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware," in *International Workshop on OpenCL*, ACM, May 10, 2022, pp. 1–12. DOI: 10.1145/3529538.3529980

[38] F. Dai, M. A. Hossain, and Y. Wang, *State of the art in parallel and distributed systems: Emerging trends and challenges*, Dec. 17, 2024. DOI: 10.20944/preprints202412.1361.v1

[39] S. Gregersen, *Aneris: A mechanised logic for modular reasoning about distributed systems*, Mar. 22, 2021. DOI: 10.26226/morressier.604907f41a80aac83ca25d44

[40] K. Wolsing, L. Roepert, J. Bauer, and K. Wehrle, "Anomaly detection in maritime ais tracks: A review of recent approaches," *Journal of Marine Science and Engineering*, vol. 10, no. 1, pp. 112–112, Jan. 14, 2022. DOI: 10.3390/jmse10010112

[41] C. Bicer, I. Murturi, P. K. Donta, and S. Dustdar, "Blockchain-based zero trust on the edge," in *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, IEEE, Dec. 13, 2023, pp. 1006–1013. DOI: 10.1109/csci62032.2023.00167

[42] C. M. Zurita and F. Assis, "Hybrid control strategies for greenhouse climate regulation: Pid, fuzzy, and neuro-fuzzy comparative implementation in temperate-dry crop systems," in *2025 XV Symposium on Computing Systems Engineering (SBESC)*, IEEE, Nov. 24, 2025, pp. 1–6. DOI: 10.1109/sbesc68008.2025.11288904

[43] A. Almen and D. Dentcheva, *Fair risk optimization of distributed systems*, Sep. 6, 2025. DOI: 10.48550/arxiv.2509.05737

[44] D. Telezhenko and O. Tolstoluzka, "A conceptual model for synthesizing the architecture of virtual distributed systems," *Bulletin of V.N. Karazin Kharkiv National University, series Mathematical modeling. Information technology. Automated control systems*, no. 55, pp. 58–65, Oct. 31, 2022. DOI: 10.26565/2304-6201-2022-55-06

[45] S. L. S. Ortiz, J. G. R. Enriquez, W. M. Tan, and C. A. M. Festin, "Hotspotter: An incentivized crowdsensing system for wifi and cellular network coverage visualization," in *2024 IEEE 29th Asia Pacific Conference on Communications (APCC)*, IEEE, Nov. 5, 2024, pp. 542–548. DOI: 10.1109/apcc62576.2024.10767933

[46] H. Laamanen, *Epistemological approach to dependability of intelligent distributed systems*, Jun. 26, 2020.

[47] V. Vijaykumar, R. Chandrasekar, and T. Srinivasan, "An obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 IEEE Conference on Cybernetics and Intelligent Systems*, IEEE, 2006, pp. 1–6.

[48] M. Inoue and A. C. Badallo, *Parallel and Distributed Systems*. Feb. 1, 2019.

[49] "Distributed system model for key management," *Bulletin of TUIT: Management and Communication Technologies*, pp. 1–5, Jan. 20, 2018. DOI: 10.51348/tuitmct115

[50] F. D. Muñoz-Escoí and R. de Juan-Marín, "On synchrony in dynamic distributed systems," *Open Computer Science*, vol. 8, no. 1, pp. 154–164, Aug. 1, 2018. DOI: 10.1515/comp-2018-0014

[51] F. Durán et al., "Programming open distributed systems in maude," in *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, ACM, Sep. 9, 2024, pp. 1–12. DOI: 10.1145/3678232.3678237

[52] M. Anisetti, C. A. Ardagna, E. Damiani, and N. E. Ioini, "Certification of modern distributed systems," in Springer International Publishing, Jul. 18, 2024, pp. 41–60. DOI: 10.1007/978-3-031-59724-4_4

[53] P. S. Sapaty, "Managing distributed systems with spatial grasp patterns," in CRC Press, Feb. 27, 2024, pp. 127–145. DOI: 10.1201/9781003425267-7

[54] T. Srinivasan, R. Chandrasekar, V. Vijaykumar, V. Mahadevan, A. Meyyappan, and A. Manikandan, "Localized tree change multicast protocol for mobile ad hoc networks," in *2006 International Conference on Wireless and Mobile Communications (ICWMC'06)*, IEEE, 2006, pp. 44–44.

[55] A. Fieschi, P. Hirmer, R. Sturm, M. Eisele, and B. Mitschang, "Anonymization use cases for data transfer in the automotive domain," in *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, IEEE, Mar. 13, 2023, pp. 98–103. DOI: 10.1109/percomworkshops56833.2023.10150357

[56] H. Li, H. Liu, and O. Marin, "Sigcse - demystifying distributed systems with a storytelling method," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ACM, Mar. 3, 2021, pp. 1386–1386. DOI: 10.1145/3408877.3439702

[57] D. Lindsay, S. S. Gill, D. Smirnova, and P. Garraghan, "The evolution of distributed computing systems: From fundamental to new frontiers," *Computing*, vol. 103, no. 8, pp. 1859–1878, Jan. 30, 2021. DOI: 10.1007/s00607-020-00900-y

[58] D. Efimov, A. Polyakov, and A. Aleksandrov, *Proofs for "discretization of homogeneous systems using euler method with a state-dependent step"*, Jun. 24, 2019.

[59] A. Bolfing, "Distributed systems," in Oxford University PressOxford, Sep. 10, 2020, pp. 143–198. DOI: 10.1093/oso/9780198862840.003.0005

[60] M. Farhadi, D. Miorandi, and G. Pierre, *Blockchain enabled fog structure to provide data security in iot applications*, Jan. 15, 2019.

[61] *OSDI - Scalable Memory Protection in the PENGLAI Enclave*, Jun. 23, 2021.

[62] R. Malik, C. Ramachandran, I. Gupta, and K. Nahrstedt, "Samera: A scalable and memory-efficient feature extraction algorithm for short 3d video segments.," in *IMMERSCOM*, 2009, p. 18.

[63] M. Zaccarini et al., "Telka: Twin-enhanced learning for kubernetes applications," in *2024 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, Jun. 26, 2024, pp. 1–6. DOI: 10.1109/iscc61673.2024.10733736

[64] R. Matzutt, V. Ahlrichs, J. Pennekamp, R. Karwacik, and K. Wehrle, "A moderation framework for the swift and transparent removal of illicit blockchain content," in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, May 2, 2022, pp. 1–9. DOI: 10.1109/icbc54727.2022.9805508

[65] M. ter Beek, B. Re, M. Viroli, R. Anane, and R. Bahsoon, "Session details: Distributed systems: Ccs track," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ACM, Apr. 9, 2018. DOI: 10.1145/3258632