Horizon: A Hyper-Edge Observability Engine for Live Streaming Networks

Ziqian Liu[1], Daqian Ding[1], Rui Han[2], Shixian Guo[2], Zhendong Xie[2], Aifang Xu[2], Changqian Wang[2]
Kefei Liu[2], Jialin Li[3], Yunming Xiao[4], Heming Cui[1], Yiming Qiu[1]
[1]The University of Hong Kong [2]ByteDance [3]National University of Singapore
[4]The Chinese University of Hong Kong, Shenzhen

Abstract

Live streaming services power mainstream real-time interactions on top of dedicated live streaming networks (LiveNets). Yet making LiveNets reliable at scale is challenging: failures arise on the user-facing delivery path and within streaming protocol and application logic, so operators need both continuous *runtime monitoring* to detect and localize incidents quickly and proactive *preflight testing* to exercise changes under representative environments and sustained playback behavior. Meeting these goals hinges on the right vantage point: the observability workflow must traverse the same network paths and delivery stacks as users while remaining controllable and non-intrusive. We present Horizon, which leverages near-user, provider-managed *hyper-edge devices* and orchestrates them into a shared fleet that supports both always-on monitoring and customizable, scenario-driven validation. Horizon has been deployed in production for over three years; in 2025, it identified 2,000+ major network incidents using 100,000+ hyper-edge agents.

CCS Concepts

• **Networks** → **Network measurement**; **Social media networks**; **Error detection and error correction**; **Network monitoring**.

Keywords

live streaming networks, network monitoring, preflight testing, hyper-edge devices, network measurement

ACM Reference Format:
Ziqian Liu[1], Daqian Ding[1], Rui Han[2], Shixian Guo[2], Zhendong Xie[2], Aifang Xu[2], Changqian Wang[2] Kefei Liu[2], Jialin Li[3], Yunming Xiao[4], Heming Cui[1], Yiming Qiu[1] . 2026. Horizon: A Hyper-Edge Observability Engine for Live Streaming Networks. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829153

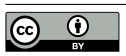
1 Introduction

Live streaming has become a mainstream medium for real-time online interaction, powering entertainment, gaming, shopping, and social experiences [1, 53, 66]. Viewers expect the stream to feel immediate and uninterrupted, so delivery must keep latency low and performance stable [37, 47]; even short disruptions can surface as stalls, quality drops, or failed sessions. Meeting this bar at scale [51] has pushed providers to build dedicated live streaming networks (LiveNets) [32, 68, 71, 72] rather than relying on generic content delivery [43, 64]. A LiveNet stitches together distributed edge sites and points of presence (PoPs) into a provider-managed fabric that carries sessions from broadcasters to viewers. It is not a passive pipe, but part of the live service itself, expected to uphold user experience at both ends of the session.

Operating a production-scale LiveNet is challenging. Each live session traverses a large, heterogeneous service backbone and, crucially, a complex near-user segment where massive traffic volumes must be served through diverse types of edge sites and last-mile network conditions. This edge-heavy heterogeneity makes LiveNet incidents frequent and varied, and many of the most damaging ones occur precisely where service-side visibility is the weakest. Moreover, LiveNets own the delivery stack end-to-end: "reachable" at lower layers does not imply that higher-layer streaming logic is correct or that media is actually playable. In production, it is common for ICMP or even regular HTTP checks to remain green while video playback fails due to streaming protocol issues. These characteristics break conventional monitoring assumptions: service-side probes do not traverse the edge-to-user paths and end-host stacks, and they cannot validate fine-grained performance signals or streaming semantics. As a result, LiveNets need *runtime monitoring* that is simultaneously user-proximate and cross-layer, so incidents can be detected where they actually manifest.

LiveNet reliability is shaped by the operating conditions the service runs under. Environment shifts and software evolution routinely expose application-level failures that do not appear in steady state. For example, frame drop rates may look benign under typical load but become user-visible during public holidays; small rollout regressions can surface as desynchronized audio and video even when network signals remain nominal. A natural strawman solution is client-side telemetry (e.g., SDK instrumentation) [6, 9, 38], which can observe rich playback signals that service-side probes cannot infer. However, it cannot catch failures before users are impacted. Because client-side telemetry is tightly coupled to real user sessions and must remain non-intrusive to user experience, it runs opportunistically and cannot stage targeted scenarios on demand. What LiveNets need is *preflight testing* that can proactively instantiate representative environments and sustained playback behavior.

Our key opportunity is a new class of near-user vantage points: large fleets of provider-managed devices deployed in homes and access networks, including set-top boxes, home gateways, and other smart devices [44, 63, 70]. We refer to them as *hyper-edge* because they sit physically closer to users than conventional edge sites, yet can be managed by service operators. This combination is powerful. Hyper-edge devices capture user-proximate network and



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829153

host conditions that datacenter and edge vantage points often miss, while avoiding the intrusiveness and unpredictability of running observability logic inside real user sessions. They also form an orchestratable substrate: operators can schedule and coordinate tasks, allocate resources, and run richer, application-aware checks than would be safe on end users, making hyper-edge devices a practical foundation for LiveNet observability. In practice, however, delivering the promise of hyper-edge devices comes with two major classes of challenges:

- **Orchestration with unreliable agents.** Hyper-edge devices operate outside datacenter-controlled conditions, so their availability and probing fidelity can fluctuate sharply over time. This makes the fleet weakly controllable: operators cannot assume any specific device is reachable and stable when a task is needed. The challenge is to extract trustworthy and timely signals from a volatile and heterogeneous fleet despite intermittent device availability and reliability, uneven capabilities, and interference between co-located probes under tight resource budgets.
- **Modeling user–environment interactions.** To catch anomalies before rollout, preflight testing must cover representative environments spanning network topology, conditions, and scale, as well as closed-loop user–application interactions where playback quality shapes user decisions and actions over time. Doing so raises questions on how to specify the intended scenario at the right abstraction level, and how to compile such specifications into coordinated executions across a geographically dispersed pool of hyper-edge devices and selected service-side nodes.

In this paper, we present Horizon, a hyper-edge observability engine with both *runtime monitoring* and *preflight testing* capabilities. For runtime monitoring, Horizon relies on a control loop that turns the unreliable hyper-edge fleet into a reliable monitoring plane. At the controller, Horizon continuously infers per-agent reliability and selects reliable agents using a confidence-driven model. Hence, monitoring results remain accurate even when a subset of devices is noisy or flaky. The controller further generates probing tasks and issues them in a pull-based manner, allowing intermittently reachable agents to acquire and relinquish tasks as their availability shifts. At each agent, Horizon characterizes assigned tasks to estimate overhead and interference, then schedules probes under tight budgets while avoiding co-locating mutually interfering tasks. Together, these designs enable timely and accurate LiveNet incident detection.

For preflight testing, Horizon introduces a programming abstraction to specify both the network environment and user behavior, together with a compilation workflow that turns each specification into a coordinated execution plan. On the environment side, Horizon realizes the specified topology and link/traffic conditions by selecting hyper-edge devices and, when needed, orchestrating selected LiveNet service nodes. On the user side, Horizon models user interactions as a signal-driven state machine: it continuously checks playback and application signals, and triggers conditional actions that drive state transitions. Horizon then compiles these environment and behavior specifications into repeatable test workflows that run across the hyper-edge fleet. Together, these designs enable systematic preflight validation.

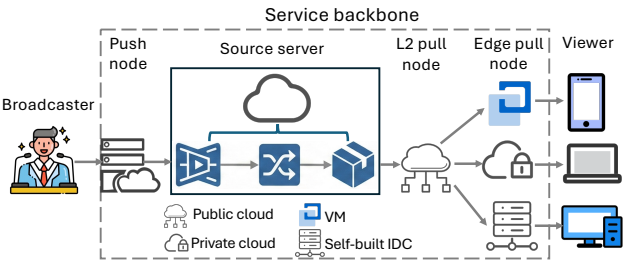


Figure 1: An end-to-end view of LiveNet architecture.

We have deployed Horizon in our LiveNet for over three years with more than 100,000 hyper-edge devices, covering all live streaming traffic passing through more than 10,000 edge nodes across over 700 sites. In 2025, the system identified more than 2,000 issues, accounting for 98% of the major incidents that were eventually confirmed. Furthermore, Horizon achieves a precision of over 95%, resulting in consistently low false alarm rates. Since its deployment, Horizon has proven indispensable to our LiveNet diagnosis teams in ensuring the reliability and stability of our live streaming service.

2 Background and Motivation

In this section, we motivate our problem further both in its real-world relevance and the choice of our techniques.

2.1 Live Streaming Network Architecture

Our LiveNet is a large-scale, edge-driven delivery infrastructure, serving billions of users globally and handling peak traffic exceeding 20 Tbps. As shown in Figure 1, it delivers a stream through a staged pipeline that is optimized for real-time propagation. Broadcasters first ingest the stream at a nearby *edge push node*, which serves as the entry point of the system. The stream is then routed to central *source servers* for transcoding and packaging into multi-bitrate renditions. These renditions are propagated to regional *level-2 (L2) pull nodes* that form a mid-tier distribution layer, and finally fetched by *edge pull nodes* that serve viewers in each geographic region. Clients retrieve media from their selected edge pull node, closing the streaming loop from broadcaster to viewer.

This delivery pipeline is inherently asymmetric. Edge push nodes are scarce and failure-critical because a single fault can affect all viewers of a stream, whereas edge pull nodes are numerous and widely distributed to absorb massive concurrency and last-mile diversity. To sustain low latency and broad coverage, LiveNets increasingly run on a hybrid deployment footprint that aggregates capacity across many providers and sites. The result is a jagged infrastructure surface: deployments span public-cloud VMs, proprietary bare-metal clusters, and self-built Internet data centers (IDCs), each with different reliability characteristics, software stacks, hardware capabilities, and operational volatility. In practice, LiveNet operators must cope not only with performance variance, but also compatibility gaps and heterogeneous failure modes that emerge from inconsistent hardware features, virtualization layers, network policies, and upgrade cadences across the hybrid infrastructure.

▹ *In summary, LiveNet infrastructure brings together a multi-stage delivery pipeline with an edge-heavy, heterogeneous deployment footprint, where small regressions can cascade into user-visible failures; this makes strong observability a core operational requirement.*

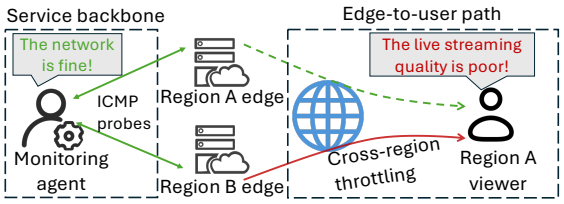


Figure 2: Cross region ISP throttling.

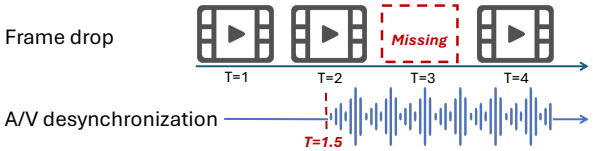


Figure 3: Frame drop and A/V desynchronization.

2.2 Observations and Requirements

LiveNet runtime monitoring. Because a LiveNet is responsible not only for moving bytes but also for sustaining correct media delivery, runtime monitoring must look beyond datacenter-visible signals and capture failures that manifest along the user-facing path and across different streaming layers. Regarding the network path, strong internal connectivity between edges does not guarantee real user reachability: the edge-to-user path goes through opaque routes or policies that can fail or degrade without any signal visible from data center or edge servers. As shown in Figure 2, we repeatedly observed *discriminatory cross-region throttling* where ISPs selectively degrade cross-boundary traffic, throttling viewers in Region A pulling streams from a Region B edge node. Since service backbone monitoring agents can still reach the Region B node without issues, these anomalies bypass server-side checks (which remain green) and prevent effective fault localization, despite the severe impact on user quality of experience (QoE).

LiveNet monitoring cannot stop at basic reachability, because streaming can fail at multiple layers of the delivery stack. Even if the backbone could confirm edge-to-user reachability, users can still experience streaming service outages or quality degradations while ICMP or even coarse HTTP status appears normal. This is because the LiveNet delivery pipeline spans multiple streaming protocols with different roles and semantics. In particular, *RTMP* [57] is primarily used by push nodes to ingest and relay live streams, while pull nodes commonly serve viewers through *FLV* [2] for lower-latency continuous delivery and *HLS* [3] for segmented delivery with broader device compatibility. As a result, probes must be explicitly streaming protocol-aware. FLV delivery is continuous, so the corresponding probes must track sustained stream progress rather than a single request outcome; HLS delivery is segmented, so probes must verify that playlists resolve to valid segment addresses and that segments can be fetched as specified.

▹ *Requirement 1: Effective monitoring must cover both the user-facing network path and the cross-layer delivery stack.*

LiveNet preflight testing. Runtime monitoring often misses failures introduced by software evolution and environment shifts, where regressions may arise above the network stack or only under specific operating conditions. Catching such failures before rollout requires exercising the live service through faithful playback behavior. As shown in Figure 3, we frequently encountered *audio–video desynchronization* in production, characterized by drifts between audio and video presentation timestamps (PTS) while all network metrics remained normal. Our analysis traced this issue to buggy transcoder updates that induced bitrate fluctuations, disrupted temporal coherence in the encoded stream, and caused synchronization failures that escaped monitoring. The same gap arises for other user-perceived anomalies, such as visual artifacts (pixelation, green screens), which can be triggered by changes in application logic or media processing and are not inferable from network signals alone.

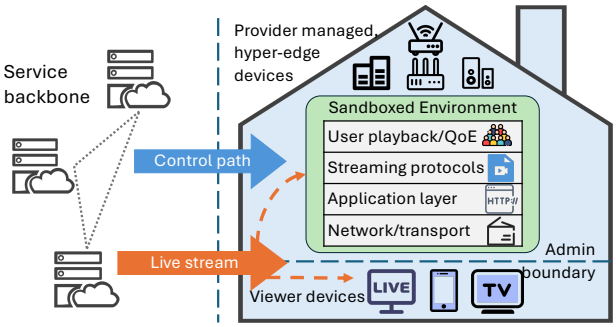


Figure 4: Hyper-edge devices as near-user vantage points.

Equally importantly, many LiveNet failures are tightly coupled with the operating environment and therefore may not surface under steady-state conditions. For example, issues that look benign during normal days can become user-visible during holiday or event spikes, when the delivery topology and traffic volume shift sharply. In particular, Figure 3 shows frame drops that repeatedly emerge only under high concurrency: when demand surges, increased contention along the delivery path can delay or lose key frames, making subsequent dependent frames undecodable and causing visible stuttering—despite the same stream appearing healthy at low volume. Preflight validation must allow operators to dynamically create test environments to evaluate whether quality of experience (QoE) remains within acceptable bounds under rare but high-impact conditions, which are not observable through monitoring alone.

▹ *Requirement 2: Effective testing must couple faithful user playback behavior with realistic operating environments.*

2.3 The Promise of Hyper-Edge Devices

In the markets where our business operates, we can leverage large fleets of managed devices deployed in homes, including set-top boxes, smart speaker, home gateways, smart fridges and other smart home devices offered and managed by third-party device providers, as illustrated in Figure 4. We refer to them as *hyper-edge devices* because they provide near-user vantage points beyond conventional edge sites. Unlike service-side probes or Client-SDK, hyper-edge devices sit close to users while providing programmable execution under operator control [56, 59, 63, 70], making them a practical foundation for LiveNet observability. In particular, they offer two core advantages:

User-proximate, cross-layer visibility. Hyper-edge devices see the network from the same vantage point as real users: their traffic flows through the identical access networks and inter-domain routes that determine user experience, capturing last-mile conditions and ISP peering impacts that existing monitoring overlooks.

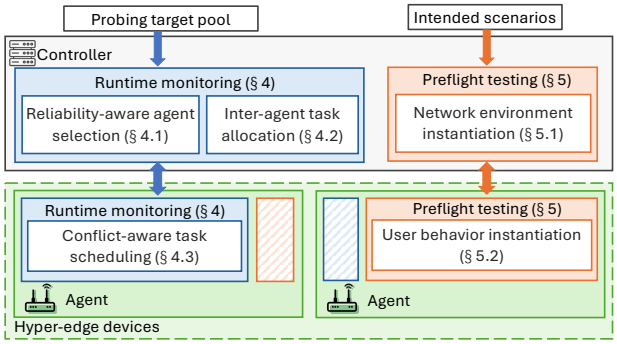


Figure 5: Overview of the Horizon system.

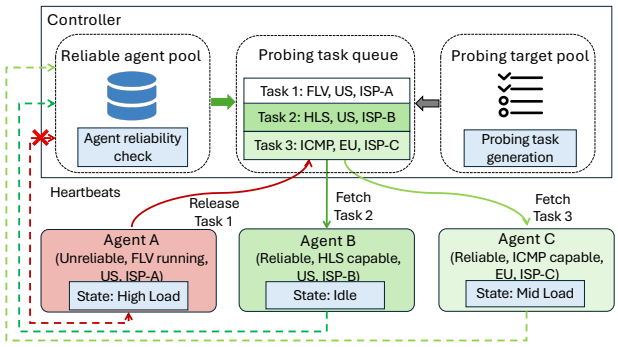


Figure 6: Control flow of runtime monitoring engine.

At the same time, these devices act as real end hosts with complete delivery stacks, enabling probes that go beyond basic reachability to exercise protocol and application semantics. This combination is critical for LiveNets, where incidents may stem from user-facing path degradation or streaming protocol-specific failures that do not surface in lower level signals. This is also where hyper-edge devices have a decisive advantage over service-side probes: without instrumenting real user clients, service-side probes rarely reach the last mile and therefore often stop at coarse reachability checks.

Behavior and environment modeling. Hyper-edge devices expose sandboxed execution environments that can run user-facing tasks, enabling the service to model how users experience live streams without intruding on real user sessions. When coordinated across many devices, they can also model diverse operating environments, spanning viewer distributions, interaction topologies, network conditions, software status, and load regimes. Together, these capabilities support systematic preflight validation and what-if analysis by exercising user–environment combinations under which failures often emerge. This is also why hyper-edge devices are preferable to client-side SDK telemetry: the latter can only observe what happens inside live user sessions, so they provide opportunistic measurements but cannot specify or replay intended scenarios—i.e., they cannot drive playback logic or instantiate controlled environments on demand.

3 System Overview

We present Horizon, a LiveNet observability engine that supports both runtime monitoring (§4) and preflight testing (§5) on top of a shared controller–agent architecture (Figure 5). A centralized controller runs within our datacenter servers, maintains global state about the hyper-edge fleet, and turns operator inputs into executable tasks; distributed agents run in sandboxed environments on hyper-edge devices and execute assigned tasks while reporting outcomes and local status. This shared infrastructure aims to address two different sets of challenges: runtime monitoring is scale-driven and constrained by reliability, whereas preflight testing is scenario-driven and constrained by expressiveness.

Horizon controller. At the controller, runtime monitoring activates two key functions. Reliability-aware agent selection (§4.1) continuously infers per-agent reliability and filters out unreliable ones, then aggregate signals across agents so the results reflect network conditions accurately. It then allocates probing tasks in a pull-based manner (§4.2): agents fetch tasks when reachable and capable, and relinquish tasks as their availability shifts, allowing the controller to make the best use of scarce reliable capacity. For preflight testing, the controller instead focuses on scenario materialization: it maps intended scenarios to concrete deployments by selecting an appropriate set of hyper-edge agents and, when needed, instrumented service-side nodes, then orchestrate them to launch coordinated execution plans (§5.1).

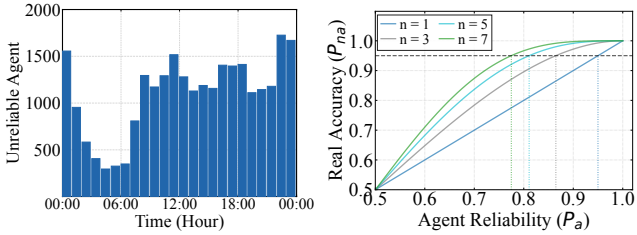
Horizon agent. Each hyper-edge device runs a Horizon agent that can switch between different modes. For runtime monitoring, the agent runs a conflict-aware scheduler (§4.3) that avoids co-locating mutually interfering probes and budgets local resources, improving both signal fidelity and throughput. For preflight testing, the agent executes compiled plans that model user behaviors (§5), where playback progresses through application states and triggers conditional actions in response to observations. Together, this architecture lets Horizon drive both runtime monitoring and preflight testing over the same fleet, with different controller and agent functions activated in each mode.

4 Runtime Monitoring Engine

As shown in Figure 6, the runtime monitoring engine of Horizon runs a closed-loop workflow between a centralized controller and hyper-edge agents. The controller converts operator-selected probing targets into a queue of probing tasks annotated with the desired user context (e.g., region and ISP) and protocol, while continuously tracking agent reliability to maintain a pool of trustworthy agents. Agents periodically send a heartbeat signal to report their status and retrieve appropriate tasks; reliable agents fetch and execute probes under local interference constraints, and release tasks back to the queue when they become overloaded or unreliable for reassignment. This workflow sustains continuous, user-proximate monitoring while remaining robust to device churn and noisy agents.

4.1 Reliability-Aware Agent Selection

Hyper-edge devices are inherently unreliable: their probes can be corrupted by transient faults, and their accuracy can drift over time. Unlike datacenter servers, these devices run in home and access-network environments where availability and measurement fidelity are affected by various factors, such as device sleep or offline periods, Wi-Fi or access-link instability, IP/ISP/geographic drift, and heterogeneous hardware capabilities. As a result, Horizon cannot treat every agent report as ground truth, even when the target service itself is healthy. We model each agent a with a reliability



(a) Reliability fluctuation **(b) Reliability v.s. accuracy**

Figure 7: Reliability of hyper-edge agents.

parameter P_a, defined as the probability that the agent produces a trustworthy probing outcome at a given time.

Reliability score estimation. Since P_a is not directly observable in production, Horizon estimates it with a windowed score computed from the agent's own probe reports against a set of reference targets. Horizon periodically probes reachability from each agent to a set of endpoints in our service backbone. These reference targets are chosen to be well-provisioned and geo-distributed so that widespread, simultaneous unreachability across many of them is rare under normal operation and would be corroborated by independent alarms. As a result, when individual agent reports an unusually high fraction of reference targets as unreachable, we treat the observation as evidence of agent-side unreliability rather than a true backbone outage.

Specifically, at each timestamp t, Horizon computes $\phi_{a,t}$, the fraction of reference targets that agent a reports as unreachable. If $\phi_{a,t}$ exceeds a threshold θ, we flag the agent's output at t as unreliable; otherwise it is considered reliable. The agent's reliability score is then the fraction of timestamps in a sliding window W that are not flagged, which serves as an estimate of P_a:

$$\widehat{P}_a = \frac{1}{|W|} \sum_{t \in W} \mathbb{I}(\phi_{a,t} < \theta) .$$

For simplicity, we use P_a to denote this estimated reliability score when the meaning is clear from context. Figure 7a further shows why P_a must be tracked continuously: agents swing from reliable to unreliable over time, with up to thousands falling below $P_a = 80\%$ within each hour. This volatility makes single-agent probing brittle and motivates two design choices in Horizon: (i) continuously selecting agents based on their current P_a, and (ii) adding redundancy by assigning multiple agents to probe the same target and aggregating their outcomes. We next quantify how redundancy turns per-agent reliability into task-level probing accuracy.

Agent selection and isolation. Concretely, consider issuing the same probing task to n agents. Let m be the number of agents whose outcomes are unreliable for this task instance. An agent produces a reliable outcome with probability P_a and an unreliable one with probability $1 - P_a$. We further assume agent failures are approximately independent: agents are distributed across different households and access networks, so local faults and access-side variability are typically not tightly correlated. Under this assumption, the probability that fewer than x outcomes are unreliable, which we treat as the task-level aggregated reliability, is:

$$P_{na}(m < x) = 1 - \sum_{i=x}^{n} \binom{n}{i} (1 - P_a)^i P_a^{n-i}$$

We set the tolerance x by the aggregation rule. Horizon uses majority voting, so an aggregated outcome is deemed reliable as long as unreliable agents do not form a majority; we therefore choose an odd n and set $x = \lceil n/2 \rceil$, making the event $m < x$ the condition that unreliable outcomes remain a strict minority. In this setting, the task-level probing accuracy after aggregation is P_{na}. Figure 7b visualizes the mapping from per-agent reliability P_a to task-level accuracy P_{na} under redundancy n. Larger n steepens the curve and lowers the required P_a, but agents with very low P_a remain unsalvageable by redundancy. Given a target accuracy P_{exp} and a chosen number of agents n, the intersection of the curve with the horizontal line $P_{na} = P_{\text{exp}}$ induces a minimum required P_a: agents below this threshold cannot meet P_{exp} even after majority aggregation, so Horizon isolates them until their reliability improves.

4.2 Inter-Agent Task Allocation

Beyond noisy measurement condition, the unreliability of the hyper-edge fleet also makes it weakly controllable: the controller cannot assume any specific hyper-edge device is reachable, stable, or able to execute on demand. To cope with this, Horizon decouples *what to probe* from *which agent runs it*, so task placement can adapt to agent churn and heterogeneous capabilities.

Probing task generation. The controller continuously updates the probing target pool using live telemetry on viewer distribution and stream popularity. It assigns each candidate target (e.g., an edge node or service endpoint) an importance score based on concurrent viewers, geographic criticality, and ISP coverage, which determines the number of agents allocated to it, encoded as tuples of the form `[target, geo, ISP, num_agents]`. The controller then expands each tuple into fine-grained probing tasks and inserts them into the global task queue Q, indexed by `(geo, ISP, protocol)` to match agents' reported context during subsequent allocation. Each task specifies the endpoint (URL/IP/port), streaming protocol (e.g., HLS/FLV), protocol-specific parameters (e.g., timeouts and duration), and the expected stream semantics needed for validation (e.g., target bitrate and codec).

Agent-initiated task pulling. Unlike stable data center and edge servers, hyper-edge devices exhibit substantial state dynamism: their reachability and resource availability fluctuate, and their effective network identity (e.g., geo/ISP inferred from public IP) can drift over time. This volatility makes push-based dispatch brittle, since the controller cannot reliably predict when a specific agent is reachable or capable. Horizon therefore adopts a *pull-based allocation strategy*: agents periodically send heartbeats to the controller, report their current profile, and pull compatible tasks when they are able.

Upon each heartbeat, the *agent* initiates a pull request to the controller. Algorithm 1 describes the controller-side handler for this request. The controller first checks the agent's reliability; if the agent is deemed unreliable, it revokes the agent's in-flight tasks and returns them to the global queue Q for reassignment, then returns empty to the agent. Otherwise, the controller responds with a set of tasks that fit the agent's advertised per-protocol budgets $A.capability$. Concretely, it looks up the matching sub-queue $Q[(A.geo, A.isp, protocol)]$ for each protocol supported by the agent, and dequeues tasks whose estimated cost fits the remaining budget, returning them as the agent's new assignment. This agent-driven

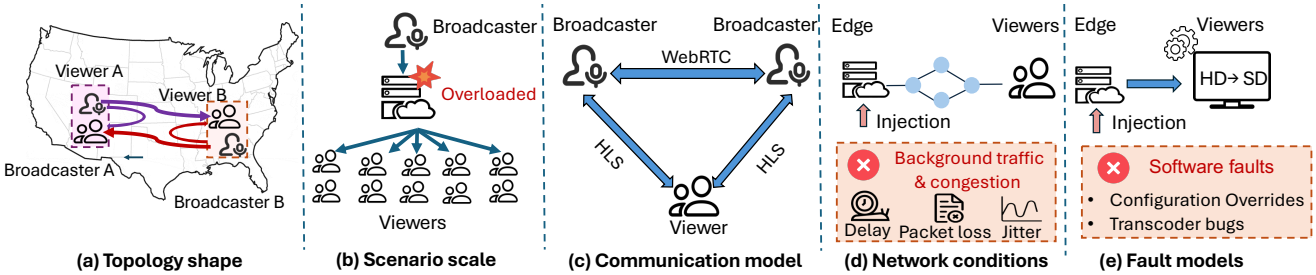


Figure 8: Examples of Horizon network environment instantiation. (a) instantiates cross-region/ISP fan-in/out structures to capture heterogeneous propagation patterns. (b) shows stress testing where a broadcaster fans out to massive viewers, overloading the intermediate node. (c) emulates a co-hosting (link-mic) where broadcasters interact via low-latency WebRTC while serving viewers via HLS, validating mixed-protocol semantics. (d) injects background traffic and link impairments to test performance under a degraded network. (e) injects transcoder bugs to verify quality switching (HD→SD) for service recovery.

Algorithm 1 Pull-Based Task Allocation

```
1:  //A denotes the agent; Q denotes the global task queue.
2:  function TaskAllocation(A, Q)
3:      // Release assigned tasks if agent is unreliable.
4:      if not ReliabilityCheck(A) then
5:          Q ← Q ∪ A.tasks;  A.tasks ← ∅
6:          return ∅
7:      // Allocate tasks under agent per-protocol budget.
8:      for (protocol, budget) ∈ A.capability do
9:          // Look up the global task queue to find tasks
10:             matching A's current (geo, isp, protocol) tuple.
11:         Q_candidate ← Q[(A.geo, A.isp, protocol)]
12:         for task in Q_candidate do
13:             cost = CostEstimation(task)
14:             // Allocate task to agent if budget is enough.
15:             if cost ≤ budget then
16:                 A.tasks ← A.tasks ∪ {task}
17:                 budget ← budget − cost
18:                 Q ← Q \ {task}
19:     return A.tasks
```

model makes task placement robust to intermittent availability while respecting tight device limits.

4.3 Conflict-Aware Task Scheduling

A Horizon agent often runs a diverse set of probes, ranging from long-lived media fetches (FLV/RTMP/HLS) to short, high-frequency network checks (ICMP/HTTP). A key challenge is that co-locating certain probe types can systematically bias each other's measurements, even when the resource capacity is not fully saturated. Horizon therefore anchors intra-agent interference on an empirically learned *conflict matrix*, and uses resource capacity as another safeguard. It then schedules tasks based on both types of constraints.

Conflict matrix construction. For each agent, Horizon periodically profiles task interference by co-running pairs of probe types under controlled conditions and checking whether one probe's reported signal deviates beyond an acceptable bound when the other is present. The outcome is a binary conflict matrix M over probe types, where $M[i, j] = 1$ indicates that running type i concurrently with type j can corrupt at least one of their measurements, and thus they should be temporally isolated during execution.

Our measurements reveal consistent, mechanism-driven conflict classes, including: (i) *queueing-induced bias*: continuous media pulls (FLV/RTMP) inflate access-link queues, causing ICMP RTT to oscillate with bufferbloat rather than reflect path latency; (ii) *connection contention*: HLS segment fetching can create many concurrent TCP connections, inflating the setup/latency observed by generic HTTP probes; (iii) *bandwidth dominance*: RTMP's persistent transmission tends to dominate the access bottleneck and amplifies jitter for other latency-sensitive probes. These conflicts are captured directly in M and provide the scheduler with an explicit notion of which probe types can safely co-run.

Capacity profiling. In parallel, Horizon enforces per-agent resource ceilings derived from pre-deployment benchmarking. Each agent is profiled to obtain a capacity ceiling C (e.g., CPU, memory, and bandwidth budget) and each probe type i is associated with an estimated resource vector r_i. This prevents the scheduler from over-committing an agent and turning benign overlap into overload-induced distortion.

Constraint-guided scheduling. Let $S(t)$ denote the set of tasks executing at time t, given a set of assigned tasks, the agent scheduler constructs an execution plan that satisfies two invariants: (1) *Capacity constraint:* at any time, the aggregate resource usage of concurrently running probes stays within the ceiling, i.e., $\sum_{i \in S(t)} r_i \le C$; (2) *Interference constraint:* probes that conflict in matrix M are not co-located in time, i.e., for any $i, j \in S(t)$, $M[i, j] = 0$. Operationally, the scheduler first places bandwidth- or connection-intensive media probes (the most common conflict sources) and then opportunistically interleaves lightweight, non-conflicting probes into the remaining slack. This strategy preserves measurement fidelity by construction while still maximizing probe throughput under tight device resource capacity budgets.

5 Preflight Testing Engine

To enable highly-customizable testing, Horizon introduces a unified domain specific language (DSL) that captures each test as a coupled specification of (i) the network environment to run under and (ii) the user playback behavior to drive through it. Operators describe both intended network environment and user behavior, while Horizon compiles the combined specification into a coordinated execution plan that can be instantiated across hyper-edge devices and selected service-side nodes, making complex preflight validation and "what-if" analysis both programmable and repeatable.

```yaml
# 1. Environment & role definition
topology:
  - role: "broadcaster"
    count: 1
    constraints:
      location: "Region_A"
      isp: "ISP_A"
  - role: "viewer"
    count: 100
    constraints:
      location: "Region_B"
      isp: "ISP_B"
      protocol: "FLV"
```

(a) Environment topology

```yaml
# 2. Function definition
functions:
  - alias: "pullStream"
    api: "@Play($config.url)"
    output: ["Status", "Content"]
  - alias: "frameGuard"
    entry_cond: "$pullStream.Status==normal"
    lambda: "@CheckFrames($pullStream.content)"
    output: ["Stall"]
  - alias: "AVSyncGuard"
    entry_cond: "$frameGuard.Stall==False"
    lambda: "@CheckPTS($pullStream.content)"
    output: ["Sync"]
```

(b) Function definition

```yaml
# 3. Execution main flows (role-specific)
workflows:
  - role: "broadcaster"
      - call: "ingestStream"
  - role: "viewer"
      - call: "pullStream"
      - wait: "10s"
      - loop: 5
          - call: "frameGuard"
          - call: "AVSyncGuard"
          - send:"$frameGuard.Stall"
          - send:"$AVSyncGuard.Sync"
          - wait: "5s"
```

(c) User behavior workflow

Figure 9: Example of Horizon scenario specification: frame drop audit in cross-region fan-out environment.

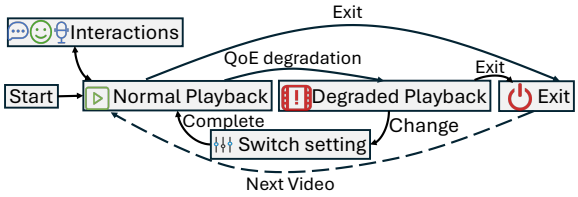


Figure 10: Example of user-application interaction

5.1 Network Environment Instantiation

We first focus on the environment side of the specification: capturing the network settings under which the live service runs, and turning them into concrete, coordinated executions. Figure 8(a–c) summarizes the core hyper-edge–driven environments that Horizon can instantiate by varying three dimensions—*topology shape*, *scenario scale*, and *communication model*. Among them, topology shape is the primary knob: Horizon instantiates cross-region/ISP fan-in and fan-out structures that capture heterogeneous propagation and aggregation patterns, and this topology serves as the scaffold for the other scenarios. On top of it, Horizon scales concurrency and traffic volume to create high-stress network environments that saturate bottleneck nodes, and varies the communication model across roles—for example, using WebRTC-like interactive sessions on broadcaster-side agents while serving viewer-side agents via HLS/FLV—to exercise realistic protocol semantics under controlled placement and load.

Next, we show that Horizon can extend environment instantiation beyond hyper-edge placement and traffic patterns by introducing two additional dimensions when needed: service-side *network conditions* and *fault models*. As shown in Figure 8(d–e) Horizon can jointly orchestrate selected service backbone nodes with the hyper-edge fleet, enabling higher-fidelity network conditions by introducing controlled background load or edge-side shaping effects (e.g., packet loss or jitter) along the delivery path, and enabling change-centric validation by applying deterministic software fault injection or configuration overrides on specific nodes. Together, these dimensions broaden environment coverage from client-side topology and load regimes to conditions that explicitly couple network dynamics with service behavior.

Building on the environment capabilities above, Horizon exposes a DSL that lets operators describe scenarios at a high level, while the controller deterministically maps them onto the hyper-edge fleet. As exemplified by the cross-region fan-out scenario in Figure 9a, the `topology` block organizes the fleet into logical functional groups via the `role` parameter (e.g., `broadcaster` vs. `viewer`). To precisely control physical allocation, `constraints` acts as a strict attribute filter, selecting eligible nodes based on network context (e.g., location, ISP) and protocol capabilities, while `count` determines the cardinality of each group. Since preflight testing is typically bottlenecked by scenario fidelity rather than scale, the controller selects and reserves a set of highly reliable matching agents for the duration of the test and uses a push-based setup to distribute role-specific configuration. It then compiles the logical topology into a running network environment by instantiating the required edges as concrete communication sessions between roles on the selected agents, configured with the specified protocol and parameters.

5.2 User Behavior Instantiation

As illustrated in Figure 10, live-stream viewers behave as a QoE-driven state machine: they continuously observe playback quality and, upon degradation (e.g., stalls, artifacts, A/V drift, low fidelity), decide whether to adjust settings (e.g., resolution, speed, toggles) or abandon the video and move on. Emulating these observe→decide→act loops is crucial for preflight testing, since many failures surface only when certain reactions are triggered and sustained over time. However, existing emulation options force a hard trade-off: pure SDK scripting efficiently replays action sequences but cannot encode QoE-conditioned decisions, while UI automation follows the real control surface but is too heavyweight and brittle to program, coordinate, and repeat across a fleet. Horizon bridges the gap with a *signal-driven execution loop* that couples lightweight QoE checks with conditional SDK-level actions, achieving closed-loop user modeling while remaining fully programmable and orchestrable.

Within the signal-driven execution loop, Horizon implements a set of lightweight, reference-free QoE checks using SDK-exposed telemetry (e.g., playback state, byte counters, and audio/video timestamps). *Frame integrity guard* flags potential decode disruption by monitoring video PTS progression: Horizon tracks the inter-frame PTS increment and marks the session as unstable when it repeatedly deviates from the expected frame period (1/FPS derived from stream metadata), indicating persistent timing irregularities consistent with corrupted playback. *Bitrate sufficiency guard* detects blurring or stuttering that is consistent with insufficient delivery rate. Horizon estimates received throughput over a sliding window of N seconds (total received bits divided by window duration) and compares it against the stream's *declared target bitrate* from metadata/manifest; sustained deficits trigger a quality-warning state. *Audio/video sync guard* tracks $|\mathrm{PTS}_A - \mathrm{PTS}_V|$ and flags persistent offsets beyond a configurable threshold (100 ms by default). Horizon uses these check outcomes as gating conditions for subsequent conditional actions (e.g., resolution switch, reconnect, or abort),

avoiding heavyweight pixel-level analysis while remaining programmable and scalable across the fleet.

Figure 9b shows how Horizon expresses user behavior logic as reusable functions. Each function wraps a user action or check, and can be guarded by an explicit `entry_cond` so checks run only when prerequisites hold. For example, `frameGuard` executes only after the stream is pulled, and `AVSyncGuard` is triggered only when `frameGuard` does not report a stall, filtering noise from transient connection failures and early-session instability. The controller then composes these functions into role-specific workflows (Figure 9c): the `broadcaster` starts the live session, while each `viewer` repeatedly pulls the stream, waits for stabilization, and invokes a bounded loop of gated checks, emitting outputs to the controller each iteration. Together with the environment specification in Figure 9a that pins network topology and scale, the behavior workflow completes an end-to-end scenario specification that Horizon can instantiate and replay deterministically across the fleet.

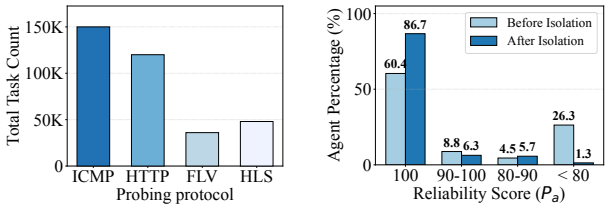
6 Deployment

We have implemented and deployed Horizon in a large-scale production environment for over three years, monitoring a global live streaming network that serves 200M+ daily active users, spans 700 edge sites, and provides an aggregate bandwidth exceeding 20 Tbps with over 10,000 edge servers (including 9,000+ pull nodes). To ensure pervasive visibility, the system orchestrates a fleet of more than 100,000 hyper-edge probing agents, covering ~90% of key business regions around the globe. The centralized controller is highly efficient, utilizing a cluster of 24 server nodes (8-core CPU, 8GB RAM) to manage this massive fleet. On the hyper-edge side, agents are built to adapt to extreme heterogeneity: we orchestrate the runtime inside a lightweight Docker container and reserve only 1 CPU core and 200MB RAM per device, which is sufficient to execute probes across diverse CPU architectures (x86-64/32, Arm64/32, MIPS64) and last-mile bandwidths (100Mbps to 1Gbps).

7 Experiment

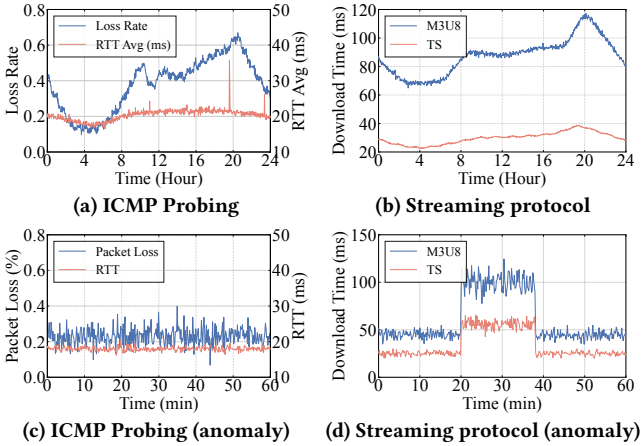
Our experimental data comprises real-time telemetry collected by Horizon's runtime monitoring engine and preflight testing engine during actual incidents over the deployment period. We establish ground truth through post-incident validation by cross-referencing Horizon alerts with confirmed customer and ISP tickets, together with subscriber-side signals such as aggregated SDK telemetry and user reports. These subscriber-side signals are often too noisy and delayed to support runtime monitoring, but they provide useful postmortem evidence for validating whether an alert corresponds to user-visible playback failures, stalls, or quality degradation.

Accuracy and effectiveness of Horizon. We evaluated the operational efficacy of Horizon using comprehensive one-year incident data in 2025. For the runtime monitoring engine, Horizon autonomously detected over 1,700 LiveNet incidents with a Mean Time to Discover (MTTD) of 3.4 minutes. Preflight testing engine has become part of our preflight validation pipeline with weekly usage, identifying over 300 bugs. A breakdown of Horizon-found issues reveals the diversity of LiveNet failures: 67% originated from edge pull nodes, 16% from L2 pull nodes, 7% from push nodes, and 10% from source nodes. Manual auditing confirmed that Horizon runtime monitoring covered over 98% of eventually confirmed



(a) Probing task distribution **(b) Agent reliability distribution**

Figure 11: Probing task and agent reliability distribution



(a) ICMP Probing **(b) Streaming protocol**

(c) ICMP Probing (anomaly) **(d) Streaming protocol (anomaly)**

Figure 12: Comprehensive analysis: Streaming 24h performance and bidirectional reachability comparison.

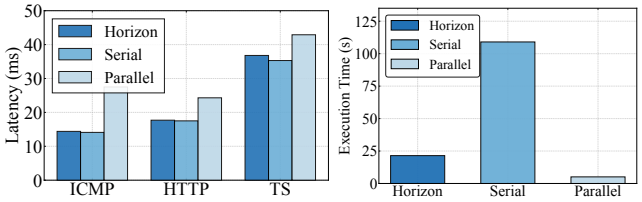
service-impacting incidents, while achieving a 95% precision rate, meaning that 95% of its reported incidents were validated as real failures. This shows that Horizon surfaces most critical incidents with few false alerts. The remaining 5% of false positives were primarily attributed to two factors: (i) inherent noise in the broadcast signals themselves, and (ii) areas with insufficient agent coverage, where the limited number of highly-reliable agents occasionally prevented adequate cross-validation of LiveNet anomalies.

Agent scalability and reliability. Figure 11a shows that Horizon robustly manages over 100,000 agents with approximately 219,000 probing tasks spanning different layers of the delivery stack. It achieves this with minimal controller overhead, averaging only 14% CPU and 7% memory usage. As shown in Figure 11b, our reliability-aware agent selection strategy significantly shifts the fleet toward higher stability: the proportion of perfectly reliable agents (scoring 100%) increased from 60.4% to 86.7%, while noise-inducing unreliable agents (scoring < 80%) plummeted from 26.3% to a negligible 1.3%. This confirms that our approach effectively filters volatile nodes to ensure trustworthy probing.

Effectiveness of multi-layered probing. We implement the streaming probes using the HLS protocol, which involves continuously fetching M3U8 playlists and MPEG-TS (Transport Stream) media segments. As shown in Figures 12a and 12b, the ICMP and streaming metrics exhibit strong temporal correlation under normal conditions, where network congestion during peak hours naturally propagates to application rebuffering. This confirms ICMP as a valid, lightweight proxy for general network performance. Conversely, Figures 12c and 12d illustrate a CPU-saturation-induced

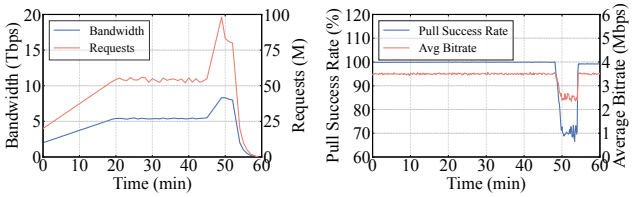
Table 1: Capabilities of Horizon preflight testing and efficiency of DSL.

Test Category	Detected Cases / Scenarios	Est. DSL Lines
Frame Integrity	Frame loss, video freezing (stall), abnormal GOP structure	~ 40
Bitrate Stability	Severe bitrate fluctuation, bandwidth throttling, encoding instability	~ 50
A/V Synchronization	Audio-video desynchronization (PTS drift > 200ms), timestamp jumps	~ 40
Bandwidth Stress	Edge node capacity saturation, throughput bottlenecks under high load	~ 60
Broadcaster Emulation	Upstream ingest instability, source-side packet loss simulation	~ 70
Disaster Recovery	Node failover latency, service continuity during forced disconnects	~ 90



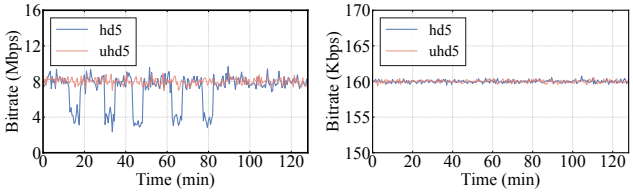
(a) Multi-layered probing result **(b) Total execution time**

Figure 13: multi-layered probe task scheduling.



(a) Bandwidth stress setting **(b) Bandwidth stress result**

Figure 14: Bandwidth stress of mock user emulation



(a) Video bitrate **(b) Audio bitrate**

Figure 15: Audio-video desynchronization behavior.

"gray failure," where kernel-handled ICMP probes remain healthy while user-space streaming suffers severe degradation due to CPU resource starvation. This divergence confirms that network-layer metrics alone leave blind spots, making application-semantic probing indispensable.

Efficiency of conflict-aware task scheduling. As shown in Figure 13, under a heterogeneous workload comprising 4 heavy TS tasks (5s each) and 40 lightweight probes (20 HTTP and 20 ICMP tasks, 0–1s each), a clear trade-off emerges between execution efficiency and measurement fidelity. Serial execution preserves fidelity by avoiding probe interference, but requires a prohibitive 109s. Parallel execution minimizes latency by launching all probes concurrently, but introduces contention-induced noise that distorts latency measurements. In contrast, Horizon's conflict-aware scheduling algorithm selectively overlaps only non-conflicting probes and isolates interference-prone tasks, achieving a 5× speedup (21.45s) while maintaining high fidelity with negligible planning overhead.

Expressiveness of DSL. Table 1 highlights the DSL's exceptional expressiveness, demonstrating how complex monitoring logic is abstracted into concise configurations. Routine validation scenarios are typically defined in just a few dozen lines, and even the most intricate disaster recovery orchestration requires fewer than 100 lines. This efficiency allows operators to rapidly deploy sophisticated checks with minimal overhead. The framework unifies frame integrity, bitrate stability, and A/V Synchronization to detect micro-anomalies like freezing and PTS drift. Furthermore, it seamlessly extends to active reliability testing, ranging from bandwidth stress to node failover, effectively transforming the fleet into a resilient, full-stack testing infrastructure.

Effectiveness of network environment emulation. We further demonstrate Horizon's value in proactive capacity planning through a bandwidth stress test conducted before a major live-streaming event. As illustrated in Figure 14a, the test emulated a flash crowd by linearly ramping up the workload to a peak of 800 Gbps and 200,000 concurrent requests, with a fixed target bitrate of 3.5 Mbps. This kind of failure is hard to uncover using existing observability tools, because neither service-side monitoring nor client-side SDK telemetry can proactively create large-scale client playback under realistic last-mile network conditions before major events occur. By contrast, Horizon can proactively generate the intended near-user workload and measure playback-level outcomes before rollout. As shown in Figure 14b, although the edge nodes initially sustained the load, a critical inflection occurred at $t = 48$ min when the workload reached its peak: the pull success rate dropped to 70%, and the actual playback bitrate fell sharply to 2.4 Mbps. The divergence between the target bitrate and the observed playback QoS revealed insufficient provisioned edge capacity, allowing operators to expand resources before the actual broadcast and avoid user-visible outages.

Effectiveness of user interaction emulation. We present an instance of audio-video desynchronization detected during a system update validation. By pre-defining playback and check logic in the DSL, Horizon first identified a persistent lag where video trailed audio by 100 ms, despite healthy metrics at the upstream ingest point. Service-side monitoring missed this case because upstream ingest remained healthy and coarse signals did not expose user-visible A/V drift. Client-side SDK telemetry can observe such symptoms only opportunistically from real sessions, rather than executing controlled pre-release playback workflows with specified duration, checks, and triggers. As illustrated in Figure 15a, this desynchronization was driven by severe video bitrate instability, with the bitrate fluctuating sharply between 8000 kbps and 2000 kbps, while the audio bitrate remained nominal and stable (Figure 15b). Correlating these findings with system telemetry further exposed abnormal memory growth and CPU saturation on the transcoder instance.

Table 2: Taxonomy of Anomalies Detected by Horizon

Domain	Failure Manifestations
Pull node	**System resource exhaustion (e.g., CPU/IO)[1]** **Discriminatory cross-region throttling[2]** **Bandwidth allocation policy verification[3]** BGP routing instabilities and DNS hijacking Edge-to-user last-mile congestion
Source node	**Transcoding and segmentation faults[4]** Authorization token denial Misconfigurations in rapid software rollouts
Push node	**RTMP authentication expiration[5]** Circular routing loops within the LiveNet Dynamic ingestion load imbalance

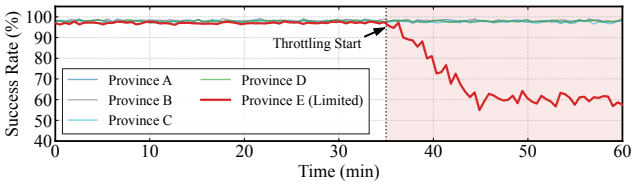


Figure 16: Cross-region ISP throttling behavior.

8 Lessons Learned

8.1 Real World Case Studies

We summarize the categories of LiveNet faults detected by Horizon in Table 2. We further present several representative, high-impact incidents that Horizon uncovered in our production environments:

Traffic spikes overwhelming edge capacity[1]. During a live streaming by a popular broadcaster, an instantaneous viewer influx caused bandwidth demand to spike by 500 Gbps. Due to an underestimation of the traffic ramp-up gradient, the scheduler allocated the stream to two under-provisioned edge nodes, each serving a distinct region and ISP, whose egress capacities were immediately saturated. Horizon runtime monitoring engine detected a sharp decline in the streaming pull success rate, dropping below 97% within seconds, and triggered an automated anomaly alert. This enabled rapid failover and proactive pull-node capacity expansion, preventing the congestion from cascading into widespread QoE degradation for viewers. Service-side monitoring alone turned out to be inadequate here, since CPU and bandwidth usage offer only an indirect indication of streaming quality, and different edge servers vary in how well they tolerate high-utilization conditions.

Discriminatory cross-region ISP throttling[2]. While 70% of users are served locally, the remaining 30% rely on cross-region delivery due to the geographic sparsity of cold-stream workloads. These cross-region sessions are particularly vulnerable to ISP policies on user-facing traffic: throttling may occur along the access or inter-domain path to user devices, while backbone and edge reachability remain healthy from the service side. To continuously monitor these long-distance user-facing paths, Horizon maintains an inter-region probing matrix across stable agents and pull nodes, tracking RTT, packet loss, and streaming pull success rates in real time. As shown in Figure 16, Horizon runtime monitoring detected a performance collapse on the path from region X to region E, where the fetch success rate dropped to ~60%, drastically deviating from the >97% stability observed on other paths (X→A/B/C/D). This pattern provides clear evidence of cross-region ISP throttling. In response, we executed a rapid reroute to local PoPs, leveraging our internal backbone to bypass throttled public segments and restore reachability via controlled intra-network paths.

Verifying multi-vendor traffic steering policies[3]. In hybrid cloud and multi-ISP architectures, traffic steering dynamically allocates bandwidth across three dimensions: region, ISP, and cloud provider. To ensure these configurations optimize the balance between delivery performance and operational costs, we utilize Horizon preflight testing to emulate diverse network environments. By deploying agents across targeted combinations of geographic and network boundaries, we obtain a "ground-truth" view of traffic distribution. This allows us to empirically validate that bandwidth steering and flow-cutting policies are executed precisely as configured, identifying potential drifts that would otherwise lead to service degradation or cost overruns.

Validating transcoding fault resilience[4]. Within LiveNet service backbone, transcoding faults that trigger visual artifacts or frame loss in a primary bitrate profile necessitate the source node to alert users to manually switch to a healthy alternative profile. We validated the effectiveness of this failover process by utilizing the Horizon preflight testing engine to inject controlled degradation into the FHD (1080p) tier. Specifically, we employed agent-side DSL logic to emulate continuous playback while monitoring for frame drops, artifacts, and audio-video desynchronization. Upon the service-side switching directive, agents transitioned to the HD (720p) backup profile, with DSL assertions confirming a QoE recovery to nominal levels in under one second, which successfully audited the system's ability to maintain service continuity despite critical upstream transcoding failures.

Push node authentication expiration[5]. In one scenario, a node reported 0% streaming success rate despite healthy infrastructure metrics (low CPU utilization, zero packet loss), confounding service backbone monitoring. Horizon runtime monitoring engine resolved this ambiguity by establishing a valid RTMP streaming connection between service backbone and hyper-edge agents, which captured repeating application-layer authentication reject events. By parsing the authentication related error code visible to the agents, we correctly attributed the failure to invalid credentials rather than network instability, enabling an automated token refresh instead of futile network troubleshooting.

8.2 Deployment Experience

Service-side monitoring attempts. Our earliest observability system relied on node-level self-monitoring inside the LiveNet service backbone: servers (e.g., source node and edge sites) reported internal health signals such as CPU, I/O, and process liveness. While useful for catching local resource and software failures, this view routinely missed network failures where nodes appeared healthy yet became unreachable. We therefore introduced inter-node mutual probing in a Pingmesh-like architecture to continuously validate service backbone reachability. This step substantially improved backbone transmission visibility, but it remained fundamentally backbone-centric: probes rarely traversed the user-facing last mile, and in practice often reduced to coarse reachability checks that could not validate streaming protocol semantics or whether the delivered media was actually playable.

Client-side telemetry attempts. To obtain user performance signals beyond reachability, we next integrated passive client-side SDK telemetry to report playback-level metrics from user sessions. This improved our ability to observe user-visible degradations, but it is ill-suited as an operational foundation for timely incident handling. SDK measurements are not always *available*: they only run when users are actively streaming and have adopted client versions that carry the required logic, which makes overall coverage and check rollout gated by user behavior and upgrade cycles. Moreover, SDK visibility is *structurally incomplete* in our setting because our LiveNet is operated as a service: it delivers a large volume of third-party live applications whose clients we cannot instrument or access at all, leaving their user-facing incidents outside the scope of our provider-oriented SDK telemetry.

Evolution of the runtime monitoring engine. To close these gaps, we moved monitoring onto the hyper-edge fleet, leveraging its user-proximate vantage points to traverse the last mile and to run protocol-aware probes that exercise the full delivery stack. In contrast to stable datacenter servers, however, hyper-edge devices are intrinsically volatile. Our first attempt was to apply statistical post-processing (e.g., smoothing and outlier filtering) to sanitize noisy results, but it could not suppress persistent noise patterns, where a small set of unstable agents repeatedly produced biased outcomes and skewed aggregates. This failure motivated a shift from post-hoc cleaning to proactive control via continuous reliability-aware agent selection. As our deployment scaled, we eventually converged to a pull-based control model based on short-lived, agent-initiated connections, rather than Pingmesh-style long-lived controller-to-agent pushes. This allows agents to pull tasks only when reachable, execute probes at coordinated times, and report results in the same control loop, while avoiding failed pushes to NATed or transiently offline devices and keeping controller instances stateless and horizontally scalable through a shared task and metadata store.

Evolution of the preflight testing engine. As LiveNet evolution accelerates, environment shifts and software updates increasingly produce failures that escape isolated lab tests. This motivated us to adopt preflight environment instantiation early on, running targeted stress tests ahead of major events and holidays. As we investigated more runtime incidents, we found that many regressions were *reaction-triggered*: they surfaced only when playback logic adapted over time (e.g., switching resolution in response to source-side signals), which environment-only tests could not exercise. Meanwhile, preflight testing quickly became a shared need across many service teams built atop our LiveNet, most of which lacked the observability team's monitoring expertise. Together, these pressures motivated our preflight testing engine and its DSL interface, which packages environment setup and user-reaction checks into reusable, repeatable tests that teams can run before rollout.

Monitoring priority. In reality, we strategically concentrate monitoring resources on critical node categories:

- *High-concurrency nodes:* Nodes serving dense user clusters act as system "heavy-hitters." Priority is assigned here to mitigate the significant blast radius of failures, which would instantly degrade QoE for massive user bases.
- *Source and push nodes:* As vital upstream linchpins, they are assigned top priority because of their large blast radius: failures can propagate through numerous downstream dependencies, triggering broad disruptions that go far beyond isolated edge issues.
- *Low-assurance infrastructure:* Nodes provided by Tier-2/3 vendors often lack reliability guarantees, requiring more granular monitoring compared to premium cloud instances.
- *Chronically unstable nodes:* Entities with a history of recurrent failures are targeted for high-frequency probing to preemptively identify regression or chronic degradation.

8.3 Discussions and Future Works

Horizon deployment model. Horizon runs on provider-managed hyper-edge devices made available through commercial partnerships with third-party device vendors, rather than on devices that we directly deploy in users' homes. As a result, coverage depends on the footprint of these partnerships and the regional density of eligible devices; regions with limited device availability naturally provide weaker observability. In practice, this limitation is operationally aligned with our service demand: regions with more viewers typically also have richer device coverage, giving Horizon stronger visibility where reliability incidents have larger impact. User consent and disclosure are handled by device providers during provisioning, including agreement on the use of anonymized measurement functionality. To isolate Horizon from normal device usage, agents run inside sandboxed environments with bounded CPU, memory, and bandwidth budgets. The sandbox also enforces data isolation: Horizon reports only operational measurement and QoE signals needed for diagnosis, and cannot inspect user traffic, user content, private application state, or household LAN activity.

Redefining edge network visibility. Conventional cloud observability focuses primarily on backbone connectivity, treating the downstream edge-to-user path as a black box that obscures faults within complex Metro Area Networks (MANs) and heterogeneous access networks. Horizon disrupts this paradigm by integrating hyper-edge devices to establish a tripartite *cloud-edge-hyperEdge* framework. This architecture extends visibility into the *Internet's capillaries*, enabling the precise detection of granular routing anomalies and last-mile micro-outages, which are critical blind spots in user-dense regions that often elude service-side monitoring.

Automating localization and diagnosis. While Horizon provides timely, user-proximate evidence of LiveNet anomalies, it does not fully automate incident localization or root-cause diagnosis. A degraded pull stream, for example, may be caused by edge resource saturation, last-mile congestion, routing changes, or service-side software faults, and operators still need to correlate probing outcomes with system telemetry, flow logs, and configuration changes. A promising next step is to reuse the preflight testing engine for differential tests: after runtime monitoring flags an incident, Horizon can generate controlled scenario variants that perturb one suspected factor at a time, then combine the results with service-side telemetry to produce ranked root-cause hypotheses.

Automating root cause analysis. We further envision an AI copilot that accelerates root-cause analysis by coupling experiment synthesis with evidence correlation. Given an incident summary and the results of differential tests, the copilot can propose the next most informative testing variants, help author and refine scenario specifications, and connect probe outcomes with service-side

telemetry to produce ranked, test-backed hypotheses. This shifts diagnosis from ad-hoc operator intuition toward a repeatable workflow where localization is driven by controlled experiments and diagnosis is guided by systematic correlation.

Diving into the DSL. Horizon leverages the DSL to compactly specify preflight scenarios and compiles them into distributed executions across the hyper-edge fleet. Operators found it useful for rollout validation and repeated preflight checks, because scenarios can be replayed consistently rather than rebuilt as one-off scripts. At the same time, different configuration or software updates often require scenario-specific validation logic, making scenario-specific authoring overhead a recurring pain point. This motivates future work on reusable templates and AI-assisted DSL generation. Beyond authoring support, the DSL also provides a structured handle on scenario behavior. Its state-machine abstraction could support richer scenarios beyond one-way live streaming, such as multi-party broadcaster–viewer interactions where users talk to each other and concurrency becomes central. This further opens opportunities for workflow validation before deployment, for example using model checking techniques [8] to detect unreachable states, missing guards, dead-end transitions, or unsafe concurrent actions in constructed scenarios.

Generalizing to other service-specific edge networks. The architectural tenets of Horizon are inherently extensible to other distributed edge services, such as WebRTC [55] and VoD [17], which similarly demand proximate-to-user and full-stack observability. Beyond swapping probe protocols, the key is to carry over the same controller–agent substrate and orchestration logic, while specializing only the service-facing semantics. Concretely, Horizon can be adapted by extending its protocol coverage and DSL with service-specific semantics for interaction workflows and experience checks, so the same monitoring/testing pipeline can operate on different service stacks.

Constructing a hyper-edge experiment gym. A promising direction is to use Horizon as an internal "experiment gym" that supports broader studies beyond incident-driven observability. By running the same scripted workloads and validation checks from many distributed user-side vantage points, we can perform apples-to-apples comparisons of competing multimedia delivery designs under matched user conditions. For example, we can benchmark different multimedia networks and stacks, and systematically vary network and security policies or protocol choices, to quantify how they impact latency, stability, and playback robustness from a true distributed-user perspective.

9 Related Works

Video streaming network. Modern live video streaming networks have evolved from traditional hierarchical CDN distribution [33, 50, 54] to flat overlay networks [26, 32], and finally toward the hyper-edge architectures [44, 63, 70], alongside protocol redesigns for real-time performance [12, 25, 34, 55, 65, 68, 72]. In parallel with these delivery optimizations, Client-SDK instrumentation has become a prevailing industry practice for runtime network observability [6, 9, 38, 40]. Approaches like Azure's media observability framework [38] employs player-embedded instrumentation [11] to collect high-fidelity QoE metrics directly from active user sessions. Horizon complements these passive, session-dependent monitoring systems by introducing a provider-managed hyper-edge infrastructure. This allows for continuous monitoring and active, controllable preflight validation that is independent of real-time user traffic.

Network monitoring and diagnosis. A large body of work has studied incident monitoring and anomaly diagnosis in both data centers [4, 5, 13–16, 18, 19, 21, 22, 24, 31, 39, 42, 46, 52, 61, 67, 74, 75] and the Internet [7, 10, 20, 23, 27–30, 35, 36, 41, 45, 48, 49, 58, 60, 62, 69, 73]. Datacenter monitoring systems such as Pingmesh [18] excel at for ensuring robust server-to-server connectivity within the internal fabric. Internet monitoring approaches, for example, BlameIt [27], employ passive diagnosis to localize faults across ISPs, and then employ traceroute-based active diagnosis to pinpoint the specific faulty AS segments. Horizon is designed to work in tandem with these datacenter and Internet monitoring systems by leveraging hyper-edge devices to explicitly probe the user-facing path, capturing the complex last-mile network conditions and upper-layer streaming protocol performance.

10 Conclusion

LiveNet observability demands both user-proximate visibility and protocol-aware, operator-controlled validation—a combination that conventional server and client side telemetry struggle to deliver in practice. This paper shows that provider-managed hyper-edge devices can bridge that gap, and demonstrates Horizon as a practical realization: a single substrate that supports runtime monitoring and preflight testing despite fleet volatility. Our production deployment indicates that this design can surface incidents quickly and at scale, substantially reducing the window of user impact.

ACKNOWLEDGMENTS

We thank the anonymous shepherd and reviewers for their insightful comments and suggestions. This work does not raise any ethical issues. Rui Han and Kefei Liu are the corresponding authors.

References

[1] 2026. Twitch. https://www.twitch.tv/. Accessed: 2026-01.

[2] Adobe Systems Incorporated. 2010. Adobe Flash Video File Format Specification. https://veovera.org/docs/legacy/video-file-format-v10-1-spec.pdf. Accessed: 2025-12.

[3] Apple Inc. 2025. Enabling Low-Latency HTTP Live Streaming (HLS). https://developer.apple.com/documentation/http-live-streaming/enabling-lowlatency-http-live-streaming-hls. Accessed: 2025-12.

[4] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 419–435.

[5] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 662–680.

[6] Bitmovin. [n. d.]. Bitmovin Player Playback Documentation. https://developer.bitmovin.com/playback/. Accessed: 2026-02.

[7] Matt Calder, Ryan Gao, Manuel Schröder, Ryan Stewart, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. 2018. Odin:{Microsoft's} scalable {Fault-Tolerant}{CDN} measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 501–517.

[8] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications.

ACM Transactions on Programming Languages and Systems (TOPLAS) 8, 2 (1986), 244–263.

[9] Conviva. 2025. Conviva Video AI Platform: QoE Monitoring and Streaming Analytics. https://docs.conviva.ai/. Accessed: 2026-02.

[10] Ítalo Cunha, Pietro Marchetta, Matt Calder, Yi-Ching Chiu, Bruno VA Machado, Antonio Pescapè, Vasileios Giotsas, Harsha V Madhyastha, and Ethan Katz-Bassett. 2016. Sibyl: a practical Internet route oracle. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 325–344.

[11] DASH Industry Forum. [n. d.]. dash.js: Reference Client Implementation for MPEG-DASH Playback. https://dashjs.org/. Accessed: 2026-02.

[12] DASH Industry Forum. 2020. Low-Latency DASH Modes. https://dashif.org/news/low-latency-dash/. Accessed: 2026-01.

[13] Rui Ding, Xunpeng Liu, Shibo Yang, Qun Huang, Baoshu Xie, Ronghua Sun, Zhi Zhang, and Bolong Cui. 2024. Rd-probe: Scalable monitoring with sufficient coverage in complex datacenter networks. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 258–273.

[14] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 217–232.

[15] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 217–232.

[16] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2019. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 549–564.

[17] Google Cloud. 2025. Video on Demand. https://cloud.google.com/use-cases/video-on-demand. Accessed: 2025-01.

[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 139–152.

[19] Shixian Guo, Kefei Liu, Yulin Lai, Yangyang Bai, Ziwei Zhao, Songlin Liu, Jianghang Ning, Gen Li, Jianwei Hu, Yongbin Dong, et al. 2025. ByteTracker: An Agentless and Real-time Path-aware Network Probing System. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 541–553.

[20] Shixian Guo, Ziqian Liu, Yangyang Bai, Yuan Chen, Kefei Liu, Qi Zhang, Songlin Liu, Yang Lv, Jianwei Hu, Gen Li, et al. 2026. Skyline: A Cloud Centric Internet Monitoring Engine. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 685–699.

[21] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.

[22] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 71–85.

[23] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.

[24] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 404–421.

[25] Xiangjie Huang, Jiayang Xu, Haiping Wang, Hebin Yu, Sandesh Dhawaskar Sathyanarayana, Shu Shi, and Zili Meng. 2025. ACE: Sending Burstiness Control for High-Quality Real-time Communication. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 1182–1198.

[26] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, et al. 2016. Via: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 286–299.

[27] Yuchen Jin, Sundararajan Renganathan, Ganesh Ananthanarayanan, Junchen Jiang, Venkata N Padmanabhan, Manuel Schroder, Matt Calder, and Arvind Krishnamurthy. 2019. Zooming in on wide-area latencies to a global cloud provider. In *Proceedings of the ACM special interest group on data communication*. 104–116.

[28] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. 2009. Moving beyond end-to-end path information to optimize CDN performance. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. 190–201.

[29] Anukool Lakhina, Mark Crovella, and Christiphe Diot. 2004. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. 201–206.

[30] Raul Landa, Lorenzo Saino, Lennert Buytenhek, and João Taveira Araújo. 2021. Staying alive: Connection path reselection at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 233–251.

[31] Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. 2023. Direct telemetry access. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 832–849.

[32] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, et al. 2022. Livenet: a low-latency video transport network for large-scale live streaming. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 812–825.

[33] Yuheng Li, Yiping Zhang, and Ruixi Yuan. 2011. Measurement and analysis of a large scale commercial mobile internet tv system. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. 209–224.

[34] Zhejiayu Ma, Frédéric Giroire, Guillaume Urvoy-Keller, and Soufiane Rouibia. 2025. Neighbor Selection Strategies in the Wild for CDN/V2V WebRTC Live Streaming: Can we learn what a good neighbor is? *Computer Networks* (2025), 111783.

[35] Harsha V Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. 2006. iPlane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 367–380.

[36] Ajay Anil Mahimkar, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Qi Zhao. 2009. Towards automated performance diagnosis in a large IPTV network. *ACM SIGCOMM Computer Communication Review* 39, 4 (2009), 231–242.

[37] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. 2022. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 193–206.

[38] Microsoft Azure. [n. d.]. Real-Time Monitoring and Observability Architecture for Media Streaming Systems. https://github.com/Azure-Samples/real-time-monitoring-and-observability-for-media. GitHub repository; Accessed: 2026-02.

[39] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 129–143.

[40] Mux Inc. 2026. Mux Data: Quality of Experience (QoE) Monitoring. https://mux.com/data. Accessed: 2026-02.

[41] Venkata N Padmanabhan, Sriram Ramabhadran, and Jitendra Padhye. 2005. Netprofiler: Profiling wide-area networks using peer cooperation. In *International Workshop on Peer-to-Peer Systems*. Springer, 80–92.

[42] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. 2017. {deTector}: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 55–68.

[43] Thomas Plagemann, Vera Goebel, Andreas Mauthe, Laurent Mathy, Thierry Turletti, and Guillaume Urvoy-Keller. 2006. From content distribution networks to content networks—issues and challenges. *Computer Communications* 29, 5 (2006), 551–562.

[44] Chunyu Qiao, Tong Liu, Yucheng Zhang, Zhiwei Fan, Pengjin Xie, Zhen Wang, and Liang Liu. 2025. PIRA: Pan-CDN Intra-video Resource Adaptation for Short Video Streaming. In *Proceedings of the 33rd ACM International Conference on Multimedia*. 11987–11995.

[45] Lin Quan, John Heidemann, and Yuri Pradkin. 2013. Trinocular: Understanding internet reliability through adaptive probing. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 255–266.

[46] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 407–418.

[47] Michael Rudow, Francis Y Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and KV Rashmi. 2023. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 953–971.

[48] Brandon Schlinker, Italo Cunha, Yi-Ching Chiu, Srikanth Sundaresan, and Ethan Katz-Bassett. 2019. Internet performance from facebook's edge. In *Proceedings of the Internet Measurement Conference*. 179–194.

[49] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi

Zeng. 2017. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 418–431.

[50] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. 2004. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. 41–54.

[51] StreamsCharts. 2026. Q4 2025 Global Livestreaming Landscape. https://streamscharts.com/news/q4-2025-global-livestreaming-landscape. Accessed: 2026-01.

[52] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. {NetBouncer}: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 599–614.

[53] TikTok. 2026. TikTok. https://www.tiktok.com/. Accessed: 2026-01.

[54] Bolun Wang, Xinyi Zhang, Gang Wang, Haitao Zheng, and Ben Y Zhao. 2016. Anatomy of a personalized livestreaming system. In *Proceedings of the 2016 Internet Measurement Conference*. 485–498.

[55] WebRTC. 2025. WebRTC Architecture. https://webrtc.github.io/webrtc-org/architecture/. Accessed: 2026-01.

[56] Dehui Wei, Jiao Zhang, Haozhe Li, Rui Han, Zhichen Xue, Yajie Peng, Xiaofei Pang, Yan Ma, and Jialin Li. 2026. {HyperEdge}: An Edge {CDN} Infrastructure for Cost Efficient Video Streaming. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2635–2650.

[57] Wikipedia contributors. 2026. Real-Time Messaging Protocol (RTMP). https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol.

[58] Yunming Xiao, Xijun Luo, Youliang Jiang, Aike Wang, Hu Chen, Zhibin Zhou, Heng Yu, Yong Jiang, Jilong Wang, Yan Chen, and Congcong Miao. 2026. DDoS Detection at the Scale of One Hundred Tbps. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026, Philadelphia, PA, USA, May 4-6, 2026*. USENIX Association.

[59] Yunming Xiao, Matteo Varvello, and Aleksandar Kuzmanovic. 2022. Monetizing spare bandwidth: The case of distributed vpns. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–27.

[60] Kaicheng Yang, Yuanpeng Li, Sheng Long, Tong Yang, Ruijie Miao, Yikai Zhao, Chaoyang Ji, Penghui Mi, Guodong Yang, Qiong Xie, et al. 2023. {AAsclepius}: Monitoring, diagnosing, and detouring at the internet peering edge. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 655–671.

[61] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 561–575.

[62] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. 2017. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 432–445.

[63] Ziwen Ye, Qing Li, Chunyu Qiao, Xiaoteng Ma, Yong Jiang, Qian Ma, Shengbin Meng, Zhenhui Yuan, and Zili Meng. 2024. {KEPC-Push}: A {Knowledge-Enhanced} Proactive Content Push Strategy for {Edge-Assisted} Video Feed Streaming. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 321–338.

[64] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. 2009. Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with LiveSky. In *Proceedings of the 17th ACM international conference on Multimedia*. 25–34.

[65] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 ACM conference on special interest group on data communication*. 325–338.

[66] YouTube. 2026. YouTube Live. https://www.youtube.com/@live. Accessed: 2026-01.

[67] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. {dShark}: A general, easy to program and scalable framework for analyzing in-network packet traces. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 207–220.

[68] Huanhuan Zhang, Congkai An, Anfu Zhou, Yifan Zhu, Weilin Sun, Yixuan Lu, Jiahao Chen, Liang Liu, Huadong Ma, and Aiguo Fei. 2024. Venus: Enhancing QoE of Crowdsourced Live Video Streaming by Exploiting Multiflow Viewer Assistance. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 170–184.

[69] Ming Zhang, Chi Zhang, Vivek S Pai, Larry L Peterson, and Randolph Y Wang. 2004. Planetseer: Internet path failure monitoring and characterization in wide-area services.. In *OSDI*, Vol. 4. 12–12.

[70] Rui-Xiao Zhang, Haiping Wang, Shu Shi, Xiaofei Pang, Yajie Peng, Zhichen Xue, and Jiangchuan Liu. 2024. Enhancing Resource Management of the World's Largest {PCDN} System for {On-Demand} Video Streaming. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 951–965.

[71] Rui-Xiao Zhang, Changpeng Yang, Xiaochan Wang, Tianchi Huang, Chenglei Wu, Jiangchuan Liu, and Lifeng Sun. 2022. Aggcast: Practical cost-effective scheduling for large-scale cloud-edge crowdsourced live streaming. In *Proceedings of the 30th ACM International Conference on Multimedia*. 3026–3034.

[72] Wei Zhang, Tong Meng, Xianhua Zeng, Wei Yang, Changqing Yan, Chao Li, Chenguang Li, Feng Qian, Junfeng Yang, Lei Zhang, et al. 2025. Harnessing WebRTC for Large-Scale Live Streaming. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 1199–1212.

[73] Zheng Zhang, Ming Zhang, Albert G Greenberg, Y Charlie Hu, Ratul Mahajan, and Blaine Christian. 2010. Optimizing Cost and Performance in Online Service Provider Networks.. In *NSDI*. 33–48.

[74] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. 2021. {LightGuardian}: A {full-visibility}, lightweight, in-band telemetry system using sketchlets. In *18th USENIX symposium on networked systems design and implementation (NSDI 21)*. 991–1010.

[75] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 76–89.