CONTENT DISTRIBUTION AT MEGA CONCURRENCY SCALE

Girish Gopalakrishnan Nair

Amazon Web Services, India]

ABSTRACT

The rapid pace of content creation, proliferation of larger screen devices and easy internet access have led to an unprecedented surge in global user demand for the best-quality content streaming. When it comes to live sports streaming, unpredictable traffic patterns during a game can often overwhelm even the most well-designed systems. Reasons for this being:

- The inadequate capabilities of traditional autoscalers to keep pace with a traffic spike;
- Infrastructure limitations that are apparent only at mega-scale; and
- Suboptimal system configurations.

This paper explores architectural strategies, resources allocation guidelines and cloud-native workflows that are time-tested to support mega-scale live streams. This paper also examines common failure scenarios and anti-patterns, identifies infrastructure bottlenecks and proposes fallback strategies to enhance resilience and reliability at massive concurrency scales. Additionally, it shares our experience of supporting live cricket streaming on the AWS cloud for Indian audiences, where traffic can surge by over 1 million viewers per minute and occasionally drops from millions to a few hundred within seconds.

INTRODUCTION

The rise of video streaming platforms has revolutionized the way that the world consumes media today. This modern streaming landscape is characterized by a relentless demand for scalable, high-performance applications that can accommodate millions of concurrent users while maintaining the highest-quality user experiences.

Previous research on large-scale video streaming has primarily focused on aspects such as video encoding best practices [1], edge computing techniques [2], client behaviour analysis [3], infrastructure scaling strategies [4][5], or caching mechanisms [6]. However, live streaming at massive concurrency levels necessitates a holistic approach where all components seamlessly integrate and operate in tandem. When executed together, the cumulative resource requirements, including computing power, storage, and network bandwidth, may exceed the available infrastructure capacity within a given geographic region or country. Specially, in a country like India where cricket is not just a sport, but is considered an emotion, over-the-top (OTT) platforms must overcome the challenges posed by finite infrastructure, limited bandwidth, and other constraints, to deliver content to millions of concurrent viewers. This paper presents our experience in supporting large-scale cricket matches, where we witness challenges like traffic surge from 1.5 million to over 10 million concurrent viewers within the first 10 minutes of the match commencing, and rapid viewership drops from 13 million to less than 4 million viewers within seconds due to interruptions. Notably, we facilitated the creation of a world record with 59 million concurrent viewers and numerous other significant events in India.

UNDERSTANDING SCALABILITY NEEDS

The global video streaming market size is expected to grow from USD 11.0 billion in 2023 to USD 25.5 billion by 2028 at a compound annual growth rate of 18.3% during the forecast period. [7]

Concurrent User Demand Trends

59 Million In 2023: A record peak concurrent audience of 59 million viewers tuned in for Disney Hotstar's coverage of the 2023 Cricket World Cup final, setting a new streaming record [8].

32 Million In 2023: Over 500 million watched IPL in 2023, Jio saw 32.1 million peak concurrency [9].

7 Million In 2023: FOX Sports says Super Bowl LVII delivered an average of 7 million simultaneous streams, up +18% over the 2022 Super Bowl [10].

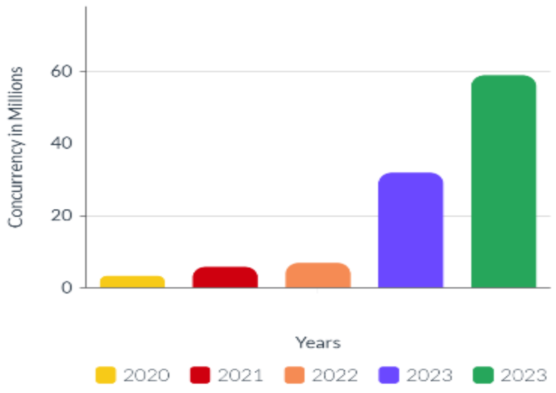

Figure 1 – Concurrency Trends

5.9 Million In 2021: The 2021 UEFA European Championship final between Italy and England peaked at 5.9 million concurrent viewers on the BBC's digital platforms (BBC, 2021).

3.4 Million In 2020: The 2020 Tokyo Olympics saw a peak of 3.4 million concurrent viewers on the NBC Sports app, a record for the network (CNBC, 2021).

TECHNICAL CHALLENGES

This section examines the practical challenges faced by streaming platforms when accommodating mega concurrent traffic. Consider a scenario where an organization aims to support a concurrency of 50 million viewers during an upcoming live event. Listed below are the challenges, potential bottlenecks and practical limitations that may arise when attempting to accommodate such a substantial concurrent user base.

Finite Infrastructure

Facilitating a massive concurrent user base demands a robust infrastructure capable of handling high traffic volumes while ensuring acceptable response times. Accommodating 50 million concurrent users necessitates substantial computational resources, upwards of thousands of CPU cores to ensure a seamless user experience. However, expecting an unlimited supply of CPUs from any infrastructure provider is unrealistic

Limited Bandwidth

Live streaming events that scale up to a concurrency of approximately 50 million viewers end up consuming 40+Tbps of bandwidth on Content Delivery Networks (CDNs) for video delivery (including 4K videos) across India. Therefore, the assumption that a single CDN possesses the requisite bandwidth capacity, represents a critical oversight, necessitating careful consideration in planning delivery strategies.

ISP Peering

Customer diversity, encompassing geographical locations, devices and network providers, are critical factors in effective traffic distribution planning. If a substantial portion of the traffic is expected from a single Internet Service Provider (ISP), it may overwhelm the peering junction between the CDN and ISP. Such scenarios cannot be effectively simulated during testing.

Sudden Traffic Fluctuation

The occurrence of traffic spikes and drops are inevitable, presuming traditional cluster auto-scaling mechanisms are adept at managing such fluctuation presents a notable concern.

SCALING STRATEGIES AND GUIDELINES

Auto-scaling offers a dynamic approach for resource provisioning, enabling organizations to adjust their computing infrastructure based on demand fluctuations. Some technical challenges highlighted by this paper can potentially be addressed by determining the optimal timing for scaling actions (when to scale) and the appropriate magnitude of resource adjustments (how much to scale).

Scale Cube

The scale cube model talks about 3 dimensions of scaling :

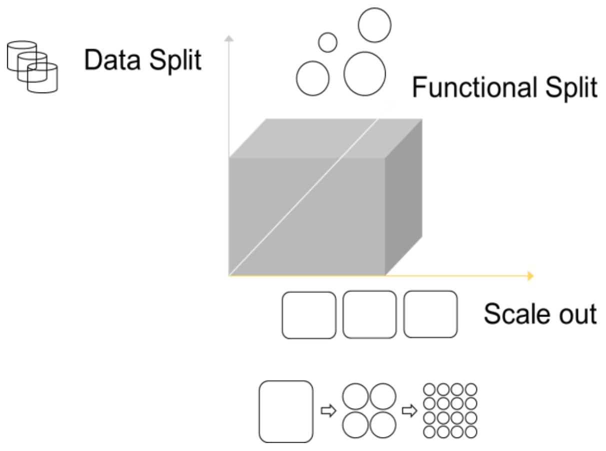

Figure 2 – Cube Scale

Scale Out

The X-axis of the scale Cube (Fig. 2) represents horizontal scaling, achieved through a technique known as scale-out. This approach involves deploying multiple instances of the application behind a load balancer. The load balancer distributes incoming requests among these application instances based on a chosen algorithm (e.g., Round Robin, Least Outstanding Requests). However, a critical challenge associated with this approach is data access. Each application instance may require access to the entire data set, necessitating a large cache to maintain acceptable performance under high concurrency loads.

Data Split

The Y-axis of the scale cube (Fig. 2) denotes data partitioning. This strategy often involves deploying micro services, with each service encapsulating a specific functionality and having access to its relevant data subset. Decomposition techniques such as verb-based or noun-based approaches can be employed to achieve this data partitioning. Combined with horizontal scaling (scale-out), this approach offers a robust scaling strategy for large-scale applications. Additionally, it serves as a fundamental unit for load estimation and subsequent scaling decisions. Hereafter, we will refer to this unit as a scaling domain for consistency throughout the paper.

Functional Split

The Z-axis of the scale cube (Fig. 2) represents functional partitioning. This approach leverages scaling by deploying multiple, identical application instances. However, each instance is assigned a specific data subset, ensuring data isolation. A dedicated routing component (e.g., load balancer or CDN) directs incoming requests to the appropriate cluster based on pre-defined criteria. Common routing strategies include utilizing request attributes like the primary key of the accessed entity or customer segmentation (e.g. routing requests from paying customers with higher Service Level Agreements (SLAs) to servers with enhanced capacity). This approach offers a significant advantage in failure isolation, as a malfunction within a single instance only impacts its associated data subset. When combined with other scaling techniques, functional partitioning contributes to enhanced platform scalability and reliability.

Cap Theorem

The **C,** A, P in CAP theorem stand for:

1. C (Consistency): Every node in a distributed cluster returns the same, most recent response after a successful write operation.
2. A (Availability): Every read or write request for a data item results in either a successful completion or a clear error message indicating the failure.
3. P (Partition Tolerance): The system continues to operate even when the network connecting the nodes experiences a failure that partitions the network. Nodes within each partition can still communicate with each other, but communication across partitions is disrupted.

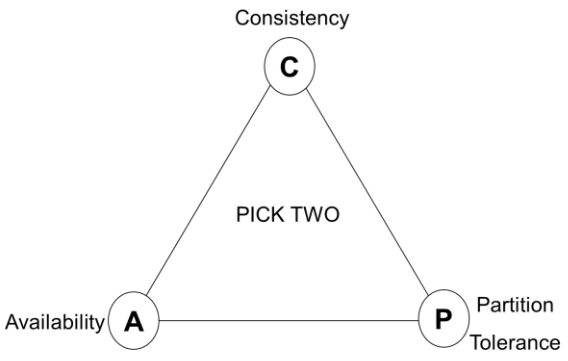

Figure 3 – Cap Theorem

In a distributed streaming ecosystem, achieving CAP simultaneously is not possible. Therefore, a clear comprehension of when and what to sacrifice, as well as the appropriate extent of such trade-offs, is a critical factor in facilitating the adoption of the scaling practices delineated within this paper.

Algorithm: Concurrent-Viewer-based-Dynamic-Scaling

Having established the principal guidelines for building scalable streaming services, the paper proposes an algorithm to facilitate scaling beyond these spiky traffic patterns. Traditional cluster auto-scaling approaches rely on metrics such as **CPU** and **memory utilization**. While these auto-scalers can be effectively utilized for non-critical (non-P0) services or day-to-day traffic, but they are inadequate for scaling P0 services during spiky traffic situations, as they do not guarantee the completion of scaling before a traffic surge overwhelms the available server capacity. The paper evaluates the use of a "**Concurrent Viewers Metric**" (C_{vm}) as a key parameter to guide the auto-scaling mechanisms of a streaming platform.

The algorithm can effectively guide the scaler to auto-scale down but the approach for scaling down the infrastructure requires more careful consideration. Automatic triggers for downscaling based solely on a reduction in traffic can be risky, especially at massive scale, due to the possibility of false positives reported by the various sources. In most practical scenarios, a manual trigger is preferred to initiate the downscaling process. This manual approach is typically employed after the conclusion of the live event or when a sustained decrease in viewership is confirmed, rather than relying on immediate traffic reduction signals.

```
1: Variables:
    $V_a$: Current number of actively viewing users
    $V_n$: Number of newly logged-in users
    $C_{vm}$: Concurrent Viewers Metric, calculated as $C_{vm} = V_a + V_n$
    $V_i$: Target initial traffic volume for the event
    $V_l$: Ramp-up ladder of new viewers
    $V_b$: Buffer capacity to accommodate new viewers during scaling
    $V_t$: Scaling trigger threshold
    $S_f$: Scaling factor, metric which will be used to determine the instances to be added

 2: Initialize variables defined:
        $V_a = C_{vm} = V_n = V_t = 0$
        $V_i$, $V_l$, Vb = pre-configured values

3: During the event, update the variables in each time interval:
        Measure $V_a$ (current actively viewing users)
          $V_a = Max(S_1, S_2, \dots S_n)$ Maximum value of users detected from multiple sources.
Where:
        $S_1$ represents the sum of unique users fetching a particular content segment from
            the CDN
        $S_2$ represents a unique heartbeat sent by a mobile application
        $S_i$ represents unique user numbers obtained from multiple sources
        Measure $V_n$ (Realtime stream of logged-in users from Authentication Service)
        Calculate $\mathbf{C_{vm}} = V_a + V_n$

   4: Scaling Trigger Condition:
        **If** $C_{vm} > V_t$ **then**
            Calculate delta between time intervals of each trigger (ΔT)
                $\Delta T = T_i - T_{i-1}$
                    -   Where $T_i$ is the timestamp of the ith trigger.
                    -   $T_{i-1}$ is the timestamp of the (i−1)th trigger.
            Calculate mean of time intervals (μT)
                $\mu T = (T_0 + T_1 + \dots + T_n)/n$
                    -   Where n is the total number of triggers
                    -   $T_n$ is time of n the trigger
            Calculate deviation of time intervals of each trigger from mean
                $S_f$ = deviation $(\Delta T - \mu T)$
            Empower auto-scaler to calculate required capacity
              Trigger Karpenter Scaling($S_f$)
            Update Scaling Trigger Threshold:
               $V_t = V_t + V_l - V_b$
        This adjusts the scaling trigger to account for the expected ramp-up and the buffer
        capacity
```

HOW WE DO IT

Having established a fundamental understanding of the core principles and algorithms, it is now crucial to delve into the practical aspects of running such a system successfully in the real world. This section will provide a detailed examination of the end-to-end process, from the preparatory stages to the execution of the live event.

Planning and Prioritization

Large-scale event management necessitates meticulous planning. While there are no silver bullets to address all scaling challenges at once, effective strategies can be progressively refined through the experience gained from conducting such events over time. Cloud Service Providers (CSPs), like Amazon Web Services' IEM (Infrastructure Event Management) & MEM (Media Event Management) teams, are involved in planning and testing phases which typically ranges from 3 to 6 months for most platforms. This experience is necessary to ensure that no critical aspect is overlooked, as even minor misconfigurations or miscalculations can adversely impact streaming performance. The challenges associated with large-scale video streaming are multifaceted, including bandwidth requirements, content delivery, scalability, redundancy, and fault tolerance. The MEM team enhances the operational reliability of business-critical video streaming events. They cover a suite of AWS media services and follow a structured four-phase approach to ensure the successful delivery of large-scale video streaming events:

1. **Pre-event Review and Preparation**: The MEM team conducts an Operation Readiness Review assessment to identify and mitigate potential risks in the video streaming architecture and configuration before the event.
2. **Event Preparation and Testing**: The team develops a comprehensive end-to-end test plan, creating operational runbook guidance and weekly status updates.
3. **Live Support during the Event**: During the live event, the MEM team engages subject matter experts (SMEs) to provide effective monitoring, troubleshooting, and triaging support.
4. **Post-event Analysis, Retrospection, and Summary**: After the event, the MEM team conducts a thorough analysis and retrospection. An event summary report is generated, which includes an assessment of learning and customer engagement evaluation.

Prioritization

The first step of prioritization is giving up aspirations to run everything; even under extreme circumstances. For OTT platforms, the top priority should be ensuring continuous content playback for viewers. The user flow and potential user paths associated with achieving smooth playback becomes your P0-Workflow (Priority Zero Workflow). All services that directly enable viewers to complete this workflow and successfully view content are classified as P0 services (Priority Zero Services). The remaining services are categorized as Non-P0s. The prioritization of Non-P0 services (e.g. P1, P2, etc.) can be further refined based on their specific platform architectures. For the purposes of this paper, we will focus on the categorization of services into P0 and Non-P0 categories.

The P0 services underpinning the core user experience (P0 workflow) necessitate utmost reliability and elasticity to guarantee uninterrupted content delivery. In this context, the paper will examine a typical P0 workflow for live cricket streaming: the typical architectures employed, the specific parameters that facilitate optimal performance, and contingency measures in the event of unforeseen disruptions:

P0.1: Authentication Services

A user's interaction with the platform typically begins with the authentication process. The majority of users will be already signed-in or have an existing account, and mere authorisation will suffice. The usage patterns for the Registration API, Login API, and Authorization APIs will vary significantly, and each of these components falls under distinct scaling domains.

Reference architecture

The Authorization APIs play a crucial role in the video streaming platform, as they are invoked for every user request, regardless of whether the user is logged in or not. Every service within the platform leverages the Authorization APIs to validate the user's permissions and access rights. This high-frequency usage pattern necessitates a different architectural approach compared to the Sign-in/Sign-up APIs. A serverless set-up, backed by a NoSQL database or an in-memory cache, can be an effective solution for scaling the Authorization APIs seamlessly. This approach allows for automatic scaling based on demand, ensuring optimal performance and cost-efficiency. However, when operating at a large scale, certain parameters may require fine-tuning to maintain the desired level of performance and reliability.

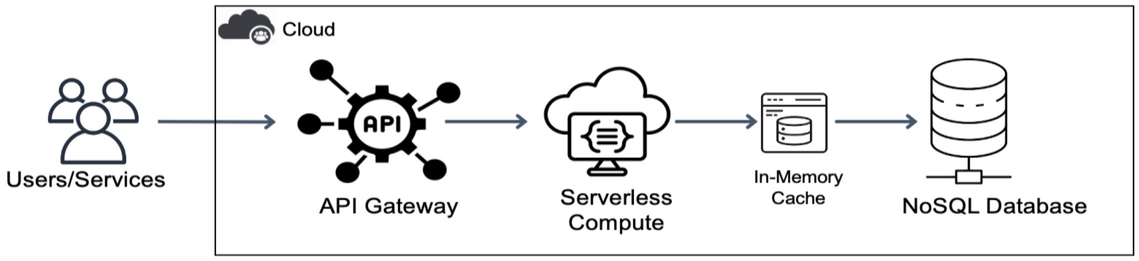

Figure 4 – Authorisation Micro-service

Parameters & Considerations

The API Gateway allows configuring burst request limits and maximum request limits. These limits need to be increased in anticipation of high-traffic events. However when using an Amazon API Gateway, an often overlooked aspect is the ability to customize the Integration Request Timeout value for API integrations. The default timeout value is set to 29 seconds, but most API calls, even under scaled traffic, typically complete within 1 second unless the backend is throttled and unable to serve requests efficiently. By reducing the timeout value to a lower value, such as 1 or 2 seconds, the API Gateway can scale seamlessly and optimally utilize resources for the given workload. For fire-and-forget APIs, like those used to capture application heartbeats, the timeout can be set even lower, around 50 milliseconds.

To handle the inrush of hundreds of thousands of users joining a live stream, the platform can engage a NoSQL database like Amazon DynamoDB. DynamoDB's "on-demand" capacity mode, coupled with a well-designed key schema can help address this inrush. Additionally, to cater to the read-heavy nature of live events, deploying an in-memory cache like DAX (DynamoDB Accelerator) can provide increased throughput and guard against increased costs due to over-provisioned read capacity in DynamoDB.

Panic Modes

To ensure optimal latency and reliability, all APIs in the video streaming platform should be accessed via a Content Delivery Network (CDN) which helps with HTTP Cloning. Apart from that, in an event of increased requests per second (RPS) (more than the planned capacity), a "panic mode" can be enabled at the CDN level. This mode will respond with cached responses, allowing all users to gain access to the system and watch the live event. This will ensures that all customers are able to stream some content at any given point in time, prioritizing availability over consistency.

API Gateways provide a feature called "Usage Keys," which can be leveraged to implement rate-limiting and throttling seamlessly. This feature helps to safeguard the backend services from being overwhelmed by excessive traffic, ensuring that the platform remains responsive and stable even during periods of high demand.

P0.2: Dashboard/Home Page

After successful authentication, users land on the Home Page, where they can: explore available content, search for live matches, and start watching them. The Dashboard page is critical and architecturally complex as it is built from responses from multiple APIs, including:

1. Geo-location Service: This API fetches the user's location to provide location-specific content and personalization.
2. Personalization Service: Based on the user's location and other preferences, this API retrieves personalized content recommendations.
3. Continue-Watching Service: This API fetches partially watched or bookmarked content for the user, enabling them to resume their viewing experience seamlessly.

When a user clicks on specific content to watch, other APIs are invoked, such as:

1. Get CDN: This API determines the optimal CDN for streaming the requested content, ensuring low latency and high-quality playback; and
2. Entitlement API: This API verifies the user's access rights and entitlements to the requested content, ensuring that authorized users can access only the content they have subscribed to.

Thus the Dashboard page is built from responses from both critical (P0) and non-critical (non-P0) APIs, which introduces complexity in terms of performance, scalability, and fault tolerance.

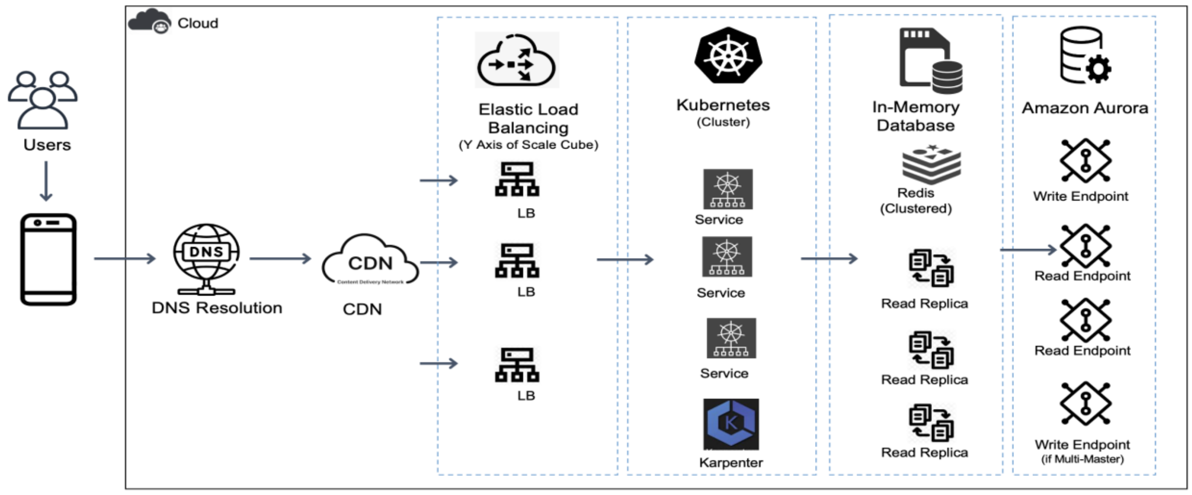

Figure 5 – Backend Services

Reference architecture

The backend APIs of the video streaming platform face significant load and stress during critical periods, such as the start and end of a live match. At the beginning of a match, a surge in traffic is expected as viewers attempt to access the platform and start streaming the content. Similarly, when the match concludes, viewers tend to click on the home page before exiting the platform, potentially generating another spike in dashboard traffic.

If this scenario is not properly addressed, it can potentially overwhelm the infrastructure and lead to system failures. While the architecture components deployed for business-as-usual (BAU) traffic may be similar, the approach to handling large-scale events requires specific modifications and optimizations.

Parameters & Considerations

Some key optimizations here that can help you scale seamlessly during spiky events are:

Load balancer Sharding and Pre-warming: This architecture has multiple load-balancers, which implement sharding and distribute traffic across multiple Application Load Balancers (ALBs). This helps to mitigate potential throttling or resource exhaustion during traffic spikes by invoking Route53's weighted routing policy to route traffic to shards. ALBs behind a CDN are also employed for improved performance and reduced latency. The load balancer is also pre-warmed, for events expecting flash traffic. Whether to shard an ALB is a decision normally taken at the load testing phase by looking at following parameters:

1. Number of requests per second expected;
2. Average size of request (Headers + body); and
3. Average response time of request.

EKS Clusters: Carefully select the appropriate instance types for Kubernetes nodes. Larger instance sizes are a no-brainer when it comes to launching instances, as it reduces the number of instances to be launched and reduces the frequency of scaling. But this can lead to CPU underutilization, particularly when using Kubernetes that limit the number of pods that can be deployed on a single instance. Kubernetes features like resource requests limits and node affinity are employed to ensure efficient resource utilization and workload distribution. High-performance Kubernetes cluster auto-scalers, such as Karpenter, are used to offload heavy processing by scaling the underlying cloud provider's compute service (like Amazon EKS) achieving optimised computing in least possible of time

Panic Modes

In scenarios where resources get constrained, disabling Non-P0 services is an effective approach. This approach will free-up Non-P0 resources allowing p0 services to consume these and scale.

Another panic strategy is to implement static responses from the API. For instance, instead of invoking backend services to construct dynamic dashboard responses for each user request, a pre-built static dashboard response can be served. This approach allows users to view essential information on the dashboard and initiate live stream playback with minimal processing overhead. This panic mode can be enabled at the Content Delivery Network (CDN) level, ensuring minimal change at infrastructure.

Another panic behaviour could be serving a default. In cases where the computation of the optimal CDN URL for a 'GetCDN' request is unavailable, a primary or preconfigured CDN can be used as a fallback option. This ensures that users can access content from any one CDN even if that is not the best one to serve content to particular user.

P0.3: Video Streaming - Playback of secured content

Upon clicking the Live Match Banner, the video playback is initiated. For a video streaming platform, the core functionality is delivering smooth, uninterrupted streams to viewers. Playback must remain flawless at all times and for all viewers, regardless of the scaling event's magnitude. A viewer's experience is not contingent upon the platform's ability to handle a specific number of concurrent streams; rather, it hinges on a near-flawless viewing experience for their chosen content.

Reference architecture

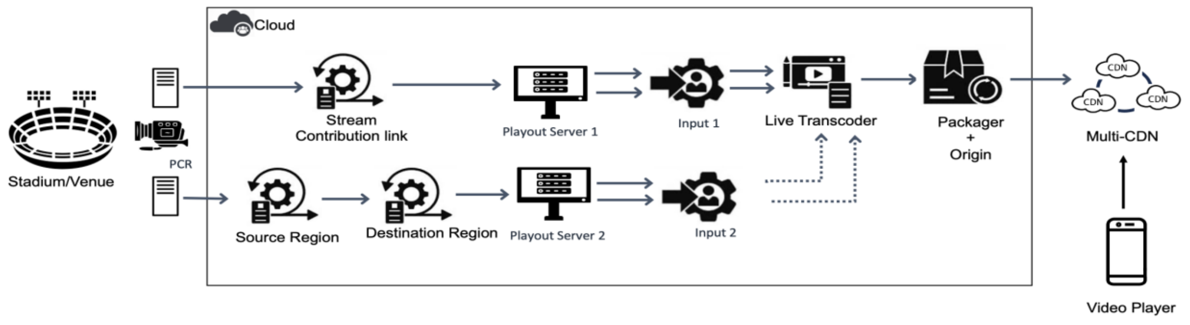

Figure 6 – Video Pipeline Architecture

The reference architecture (Figure 6) initiates requests for redundancy across all stages of the streaming pipeline – from ingestion, playout, encoder, packager to CDN. This redundancy ensures continued service availability and minimizes the impact of potential component failures.

Parameters & Considerations

Pipeline Locking

Redundancy in large-scale video streaming architectures ensures high reliability but introduces complexities in managing a seamless viewing experience. The key challenge arises from potential mismatches in chunk IDs and frame alignment across redundant delivery paths, leading to disruptions or inconsistent playback. The technique AWS deploys to mitigate this issue is MediaLive's "Pipeline Locking" feature. This feature provides frame-accurate outputs by synchronizing the video and audio pipelines.

Multi-CDN

Relying on a single CDN to serve a diverse customer base is often insufficient for achieving optimal performance. CDN performance can vary across different locations and time periods, as certain CDNs may perform better in specific regions or during certain times of the day. To address this challenge, the video streaming system should be designed to leverage the best-performing CDN for a given location and time, ensuring a superior Quality of Experience (QoE) for end-users. The intelligence in selecting the optimal CDN is derived from metrics such as Video Start-up Time (VST), Video Start-up Failures (VSF), rebuffering events and playback failures, etc. By analysing these metrics, areas where streaming performance is suboptimal and negatively impacting QoE can be identified and the root causes of these issues can be pinpointed. Utilizing multiple CDNs mitigates the risk of outages and bandwidth limitations of one CDN.

Timely corrective actions can be implemented by dynamically switching to a better-performing CDN.Streaming Protocol considerations: HLS or DASH or Both?

The DASH (Dynamic Adaptive Streaming over HTTP) manifest, is an XML-based file that contains essential information for video playback. HLS (HTTP Live Streaming) manifest is comparatively easy to read and was developed much earlier than DASH. Both protocols are efficient in delivering adaptive bit rate streaming.

When deciding on the appropriate protocol for high-concurrency events, even minor factors can play a significant role. Although DASH is defined as an industry standard, with wide device support, our experience has shown it is not consistently supported across all platforms and devices. The Indian market encompasses a diverse range of alternative operating systems, including KaiOS, iOS, and Linux distributions, which currently lack DASH compatibility.

A specific challenge observed with DASH for live events when enabling Segment Timeline is variation in audio segment sizes. The Segment Timeline helps Server-Side Ad Insertion (SSAI) services to insert advertisements, which is crucial for generating revenue during the event. But the variation in segments complicates SSAI.

Enabling a segment timeline also result in larger manifests. At scale, large manifest sizes can become problematic. Especially when these are refreshed every two seconds. This can generate significant traffic on the CDN as well. Thus the reduced complexity and comparatively smaller size of HLS manifests have proven effective in the Indian device landscape.

Panic Modes

To ensure continuous playback and mitigate the impact of potential failures, a world-feed of the live match is provided as additional input to MediaLive. This redundancy measure ensures that viewers can continue watching the match in some language, even if the primary playout servers or the clean feed ingest fails, preventing playback errors and maintaining a continued viewing experience.

Developing plans for controlled degradation such as the intentional disabling of non-critical services during critical events, serves as an effective panic strategy. For example, if the CDN experiences bandwidth exhaustion, the ability to dynamically remove high-quality renditions, such as 4K and 1080p, becomes crucial. The ‘Manifest Filtering’ feature of a Packager (MediaPackage) helps to implement this graceful degradation technique. This prioritizing of lower resolutions enables a larger number of viewers to access the content at significantly less bandwidth. Without graceful degradation, only a limited audience would be able to access the content, potentially resulting in errors or suboptimal experiences for others.

P0.4: Monetisation - Ad insertion and reporting

In the context of cricket streaming in India, where most matches are free to watch and powered by advertisements, ad insertion is a P0 for organizations to ensure revenue generation from these streams. There are multiple options available for ad insertion, Server-Side Ad-Insertion (SSAI) being one of the most common approaches that allows seamless ad insertion without disturbing the playback.

SSAI involves the integration of an ad decision server (ADS) within the video delivery workflow. The ADS determines the relevant advertisements for a particular ad slot based on factors such as viewer demographics, content metadata, and ad campaign rules. The ad insertion server then seamlessly splices the selected advertisements into the main content manifest, creating a unified stream for delivery to the viewer. Furthermore, SSAI maintains a consistent viewing experience, which is crucial for user engagement and satisfaction.

Reference architecture

To implement server-side ad insertion (SSAI) at a large scale on the AWS cloud, AWS employs AWS MediaTailor. This acts as a manifest manipulator, responsible for dynamically inserting ad segments into the video manifest files as they are delivered to viewers. MediaTailor will front-end the video delivery pipeline to perform SSAI.

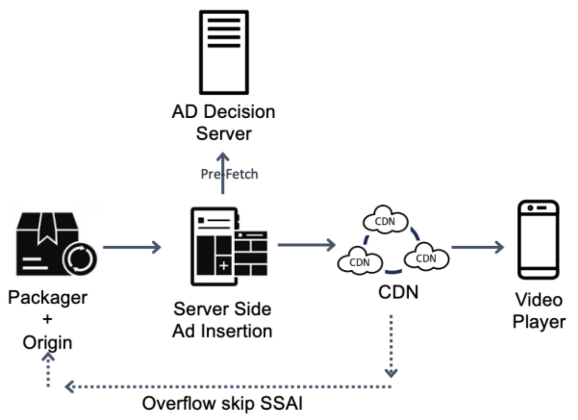

Figure 7 – SSAI Architecture

Parameters & Considerations

During large-scale live events, ad request and transcoding timeouts can lead to significant revenue leaks if not addressed properly. To mitigate this issue and maximize ad fill rates and monetization, the architecture employs ad prefetching. Ad prefetching is a proactive approach where MediaTailor fetches and transcodes ad assets in advance, before they are required for insertion. This technique serves two key purposes: it provides more time for programmatic ad trading, and it reduces ad insertion latency. Implementing ad prefetching is particularly crucial in scenarios with high concurrency, stringent latency requirements, or when dealing with a large volume of ad requests and assets. It helps to optimize ad delivery and revenue generation by minimizing the risk of timeouts and missed ad opportunities during live events with high viewer traffic. To reduce the ad transcoding time, “Transcode profile” is used which can only be configured via AWS Support as of today.

Panic Modes

Fail over and overflows

When implementing Server-Side Ad Insertion at large scales, it is crucial to consider the predefined limits of the SSAI service, as exceeding these limits can lead to request failures. To mitigate this issue, an overflow mechanism can be implemented. One approach is to configure the CDN, such as AWS CloudFront, to bypass the ad insertion process entirely when encountering errors like HTTP 429 (Too Many Requests) from the SSAI component. Alternatively, this can be achieved by configuring the GetCDN API to return the direct origin URL for the core video content, bypassing SSAI during overflow conditions. By implementing this overflow mechanism, the architecture ensures the uninterrupted delivery of core video content, even if ad insertion needs to be temporarily suspended for overflow traffic.

CONCLUSION

Successfully scaling video streaming applications to accommodate millions of concurrent viewers necessitates a multifaceted approach encompassing robust infrastructure, efficient content delivery mechanisms, effective data management strategies, and comprehensive security measures. By meticulously planning and implementing the best practices outlined throughout this paper, platform engineers can foster a resilient and scalable streaming ecosystem capable of delivering exceptional viewing experiences to a global audience.

REFERENCES

1. Layered video streaming in large-scale networks - https://www.sciencedirect.com/science/article/abs/pii/S1084804514001441
2. Large-Scale Video Stream Concurrent Transmission for Edge - https://www.mdpi.com/2227-7390/11/12/2622
3. Modelling large-scale live video streaming client behaviour - https://link.springer.com/article/10.1007/s00530-021-00788-4
4. Auto Scaling API Gateway based for K8 - https://ieeexplore.ieee.org/document/8663784
5. Proactive-Reactive Auto-Scaling Mechanism for Unpredictable Load Change - https://ieeexplore.ieee.org/document/7557733
6. Large-Scale Video Streaming in Highly Heterogeneous Environment - https://ieeexplore.ieee.org/abstract/document/4359973
7. Markets and Markets, “Video Streaming Software Market by Component (Solutions, Services)” - https://www.marketsandmarkets.com/Market-Reports/video-streaming-market-181135120.html.
8. SportsProMedia: 2023 World Cup Final - https://www.sportspromedia.com/news/cricket-world-cup-final-2023-disney-hotstar-live-streaming-viewership-record/.
9. Mint - “Over 500 million watched IPL in 2023, Jio saw 3.21 crore peak concurrency” https://www.livemint.com/industry/media/over-500-million-watched-ipl-in-2023-jio-saw-3-21-crore-peak-concurrency-11686232243972.html.

10. Steaming media blog - “thirteen years of super bowl streaming viewership stats, 2012-2024” – february 22 2024 https://www.streamingmediablog.com/2024/02/superbowl-streaming-numbers.html
11. https://blog.hotstar.com/scaling-for-tsunami-traffic-2ec290c37504
12. https://www.ibc.org/technical-papers/ibc2023-tech-papers-implementing-hls/dash-content-steering-at-scale/10258.article

ACKNOWLEDGEMENTS
The author would like to thank his colleagues for their contributions to this work. He would also like to thank the International Broadcasting Convention for permission to publish this paper.
“Mr. Tony Thomas” Head of DevOps & Quality at JIO, for valuable insights and assistance during the course of research.
“Mr. Alastair Cousins” Sr.Manager Solution Architect at AWS, for valuable insights and assistance during the course of research.
“Mr. Maheshwaran G” Principal Solution Architect at AWS, for assistance and inputs.
“Mr. Sanjay Singh” Sr Solution Architect, TMEGS at AWS Professional Services, for data points and metrics.
“Mr. Ajay Bhardwaj Swami” Sr.Technical M&E Consultant at AWS, for constructive feedback and suggestions at review.