WHITE PAPER

ACHIEVING BROADCAST-GRADE LOW LATENCY IN LIVE VIDEO STREAMING

CONTENTS

INTRODUCTION

As more and more consumers turn to streaming video to watch live events, reducing latency has become one of the most important challenges facing OTT and other online video providers. Viewers now expect the online streams of live sports, awards shows, and other tentpole events to be in sync with their broadcast counterparts. Nothing is worse than getting a notification or seeing a tweet about a crucial goal long before you see it happen on the screen.

This white paper will help readers understand exactly what latency is, what is considered low and even ultra-low latency, and how changes in the video encoding and packaging workflow can help content publishers and distribution platforms reduce latency and provide a better experience for their customers. We'll use solutions from Amazon Web Services and AWS Elemental as examples, but understand that the concepts and challenges are the same no matter what encoding and packaging service or content delivery network you are using. Let's jump right in.

WHAT IS LATENCY?

There are several different types of latency, but for our purposes, it can be defined as the time lag between when a viewer would see something via broadcast and the time he or she would see it via over-the-top (OTT) streaming on a mobile phone, tablet, computer, or connected device. As shown in **Figure 1**, even broadcast, satellite, and cable introduce some latency into the signal chain, typically somewhere around 5 seconds—meaning that there's 5 seconds from the time an event happens in a stadium or on a stage to the time a viewer sees it on screen (satellite latency runs closer to 12 seconds). As you can see under the orange arrow labeled "Streaming Latency Continuum," different online video protocols have different ranges in latency.

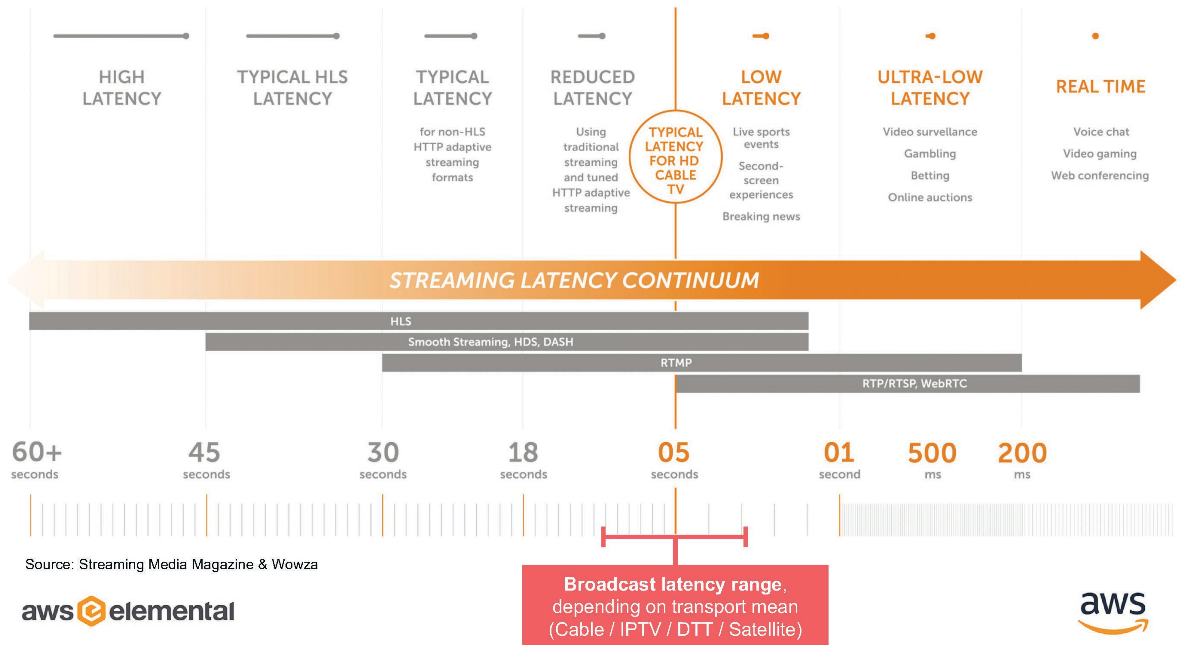

HIGH LATENCY
TYPICAL HLS LATENCY
TYPICAL LATENCY
for non-HLS HTTP adaptive streaming formats
REDUCED LATENCY
Using traditional streaming and tuned HTTP adaptive streaming
TYPICAL LATENCY FOR HD CABLE TV
LOW LATENCY
Live sports events
Second-screen experiences
Breaking news
ULTRA-LOW LATENCY
Video survellance
Gambling
Betting
Online auctions
REAL TIME
Voice chat
Video gaming
Web conferencing
STREAMING LATENCY CONTINUUM
HLS
Smooth Streaming, HDS, DASH
RTMP
RTP/RTSP, WebRTC
60+ seconds
45 seconds
30 seconds
18 seconds
05 seconds
01 second
500 ms
200 ms
Source: Streaming Media Magazine & Wowza
aws elemental
Broadcast latency range, depending on transport mean (Cable / IPTV / DTT / Satellite)
aws

Figure 1 – Latency levels, protocols, and market segments (Image courtesy of Streaming Media and Wowza)

Latency is introduced at many different points in the OTT video workflow:

- Video encoding
- Ingest and packaging
- Network propagation delays
- Transport protocol
- Content delivery network (CDN)
- Segment length (MPEG-DASH, HLS, HDS, Microsoft Smooth Streaming)
- Video player policies (buffering, playhead positioning, resilience)

We'll address each of those factors later, but it's worth looking at transport protocol and segment length now. For many years, the standard for delivering video online was Adobe Flash, which uses the RTMP transport protocol and offered latency of as little as 200 milliseconds (less than cable and broadcast!) up to 30 seconds, or 20-25 seconds behind broadcast. But Adobe announced in 2017 that it would stop updating and supporting Flash in 2020, and while there is still plenty of Flash Video online, most OTT providers have begun to move to HTTP-based protocols, most notably MPEG-DASH (Dynamic Adaptive Streaming over HTTP) and Apple's HLS (HTTP Live Streaming), but also Microsoft Smooth Streaming and Adobe HTTP Dynamic Streaming (HDS). Those are all adaptive bitrate (ABR) protocols, meaning that for any given video stream, multiple bitrates are delivered, with each viewer's bitrate dependent upon network conditions, congestion, and the type of connectivity on which the viewer is receiving the video. In order to achieve that, adaptive bitrate streams are divided into chunks or segments, so that on-the-fly changes in bitrate can be made from one bitrate to another to ensure that the viewer is always receiving the best possible quality video.

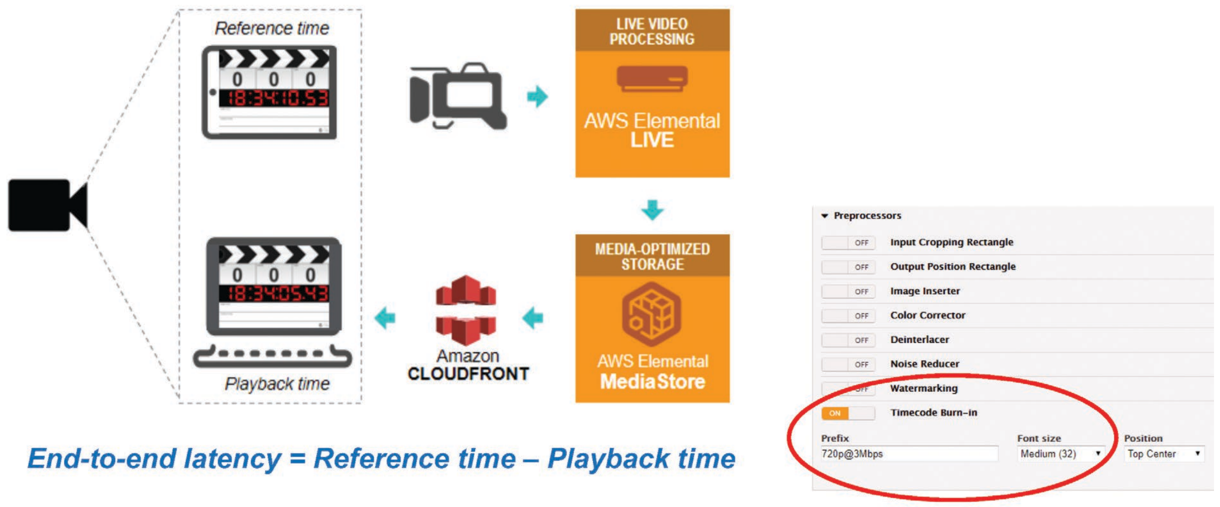
The Smooth Streaming protocol specifies 2-second segments, typically resulting in 10 seconds of latency; DASH also specifies 2-second segments, with latency usually in the range of 8 to 10 seconds. As you can see in Figure 1, latency can go as high as 45 seconds or as low as 2 seconds. Apple's guidelines for HLS specify 6-second segments for iOS apps, resulting in a typical latency of 18 to 20 seconds, though it can go as high as 60 seconds. DASH and Smooth Streaming's shorter segments are supported by most platforms, while any apps in the Apple App Store are limited by HLS's 6-second segment size.

Luckily, the Safari mobile browser in iOS 11 supports playback of shorter segments and autostart for live videos, plus FairPlay DRM support, so you can deliver high-quality, low-latency content to Apple devices directly in the browser. (Google does not specify a specific segment size for Android apps, so you have more flexibility to delivery to Android devices.)

For some applications, the WebRTC (Web Real-Time Communications) protocol is gaining traction, as it allows for lower latency than any of the delivery protocols discussed above. But WebRTC doesn't scale well, and so isn't an option for large-scale live events or OTT services. For live events going to large audiences, the goal is to get as close to broadcast and cable's 5-second latency as possible, and ideally even improve on it, using DASH or HLS.

MEASURING LATENCY

The simplest way to measure latency—and eventually determine how much each component of the workflow chain is responsible for the overall latency—is using a clapperboard application on a computer or mobile device. Then, film the clapperboard with a camera connected to an encoder, publish the stream to your origin server, and deliver to a video player via CDN. Then, put a device playing the stream next to the device running the clapperboard app, take a photo of the two screens, and subtract the timecode on the player from the timecode on the clapperboard app. The difference between those two numbers is your total latency. See **Figure 2** for a flowchart showing such a test using AWS Elemental Live for encoding, AWS Elemental MediaStore as the origin server, and Amazon CloudFront as the CDN; the same test can be performed with any encoder, origin server, and CDN.



Preprocessors

OFF	Input Cropping Rectangle
OFF	Output Position Rectangle
OFF	Image Inserter
OFF	Color Corrector
OFF	Deinterlacer
OFF	Noise Reducer
OFF	Watermarking
ON	Timecode Burn-in

Prefix	Font size	Position
720p@3Mbps	Medium (32)	Top Center

Figure 2 – Measuring end-to-end latency

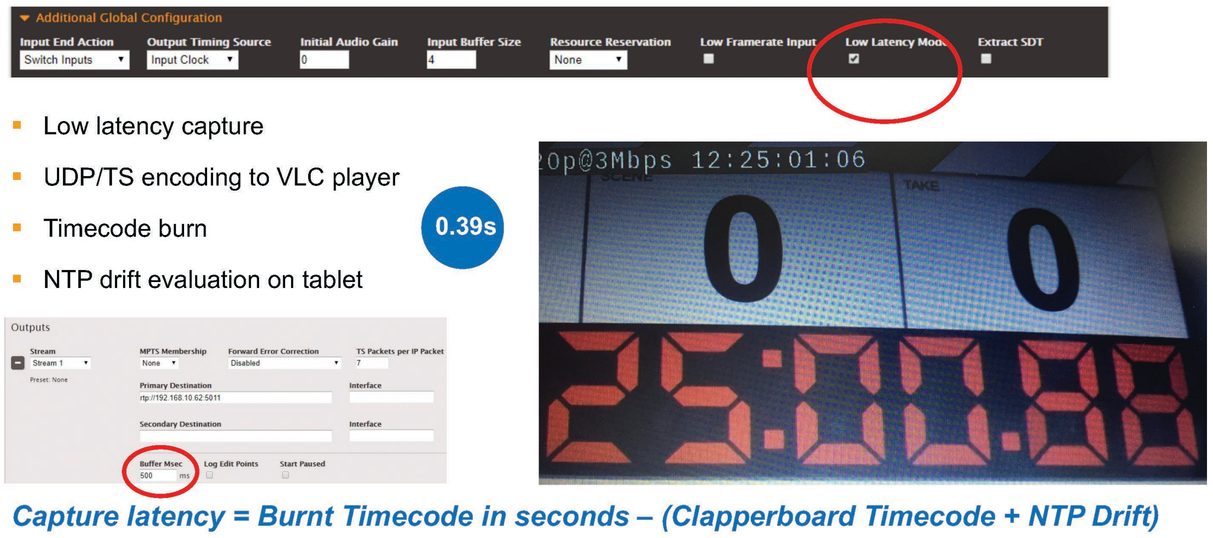
Now that you have the end-to-end latency, the next step is to determine the latency introduced at each step of the workflow, beginning with capture.

MEASURING CAPTURE LATENCY

In the Preprocessor section of AWS Elemental Live, you can activate Timecode Burn-In for each bitrate in your encoding ladder. You also need to select the low latency option in your encoder, and then set up a UDP/TS encoding event using the Network Time Protocol (NTP) with a 500-millisecond buffer and send it to a VLC player on a laptop on the same network. (You do need to account for NTP drift on the tablet, which is typically only a few milliseconds.) Then you start filming the clapperboard application, encode and send the signal to the VLC player, then take a photo of the result. Capture latency can be determined by using the following equation:

```
(burned-in timecode) – (clapperboard timecode + NTP drift) = capture latency
```

In the example shown in **Figure 3**, we see that capture introduces .39 seconds (390 milliseconds) of latency.



Additional Global Configuration

Input End Action	Output Timing Source	Initial Audio Gain	Input Buffer Size	Resource Reservation	Low Framerate Input	Low Latency Mode	Extract SDT
Switch Inputs	Input Clock	0	4	None	☐	✓	☐

- Low latency capture
- UDP/TS encoding to VLC player
- Timecode burn
- NTP drift evaluation on tablet

0.39s

Outputs

Stream	MPTS Membership	Forward Error Correction	TS Packets per IP Packet
Stream 1	None	Disabled	7

Preset: None

Primary Destination: rtp://192.168.10.62:5011
Interface:
Secondary Destination:
Interface:
Buffer Msec: 500 ms
Log Edit Points: ☐
Start Paused: ☐

Capture latency = Burnt Timecode in seconds – (Clapperboard Timecode + NTP Drift)

Figure 3 – Measuring capture latency

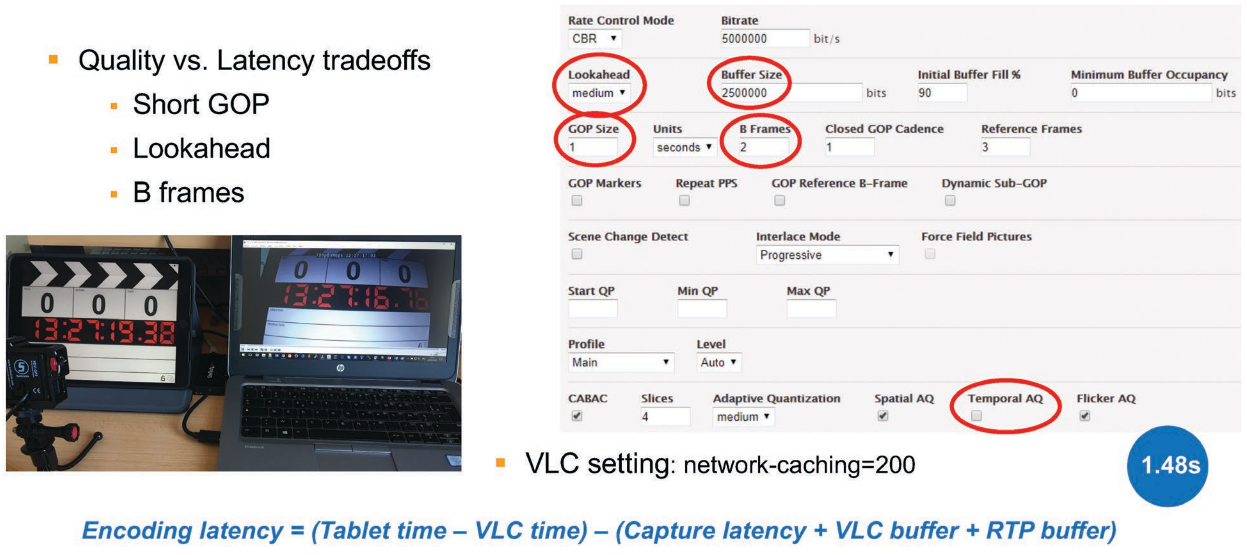
MEASURING ENCODING LATENCY

You can use the same UDP/TS event to measure encoding latency, applying the following formula:

```
tablet time – VLC time) – (capture latency + VLC buffer + RTP buffer) =
encoding latency.
```

In the example shown in **Figure 4**, the numbers are as follows:

```
(19.32-16.75) – (0.39 + 0.20 + 0.50) = 1.48 seconds encoding latency
```



- Quality vs. Latency tradeoffs
  - Short GOP
  - Lookahead
  - B frames

Rate Control Mode: CBR
Bitrate: 5000000 bit/s
Lookahead: medium
Buffer Size: 2500000 bits
Initial Buffer Fill %: 90
Minimum Buffer Occupancy: 0 bits
GOP Size: 1
Units: seconds
B Frames: 2
Closed GOP Cadence: 1
Reference Frames: 3
GOP Markers: ☐
Repeat PPS: ☐
GOP Reference B-Frame: ☐
Dynamic Sub-GOP: ☐
Scene Change Detect: ☐
Interlace Mode: Progressive
Force Field Pictures: ☐
Start QP:
Min QP:
Max QP:
Profile: Main
Level: Auto
CABAC: ✓
Slices: 4
Adaptive Quantization: medium
Spatial AQ: ✓
Temporal AQ: ☐
Flicker AQ: ✓

- VLC setting: network-caching=200

1.48s

Encoding latency = (Tablet time – VLC time) – (Capture latency + VLC buffer + RTP buffer)

Figure 4 – Measuring encoding latency

In this example, we've tweaked some of the encoding parameters to improve our latency. First, we set the Lookahead, which tells the encoder how far into the future it should look at content to determine the best quality encoding, to Medium. The more Lookahead you apply, the more latency you introduce.

Then, we set the B Frames, which tell the encoder how many frames around each B Frame need to be encoded before it can properly encode the B Frame. Again, the larger the B Frame number, the more latency will be added. In this example, we set the B Frames to 2.

The most important parameter in the encoding settings is Buffer Size (leveraged by the video decoder to build its buffer). In AWS Elemental Live, the default buffer size (which is twice the GOP size) will add 2 seconds of latency. We set the buffer size to the minimum, 2500000 for a 5Mbps encoding bitrate, which generates a half-second of latency; going any lower would compromise the quality of the decode, and our goal is to minimize latency while keeping an acceptably high level of video quality.

By tweaking these parameters, we were able to bring the encoding latency from somewhere around 5 seconds down to 1.48 seconds—a considerable improvement.

MEASURING INGEST LATENCY

Ingest latency is the time necessary to package the ingest format and to ingest it to an origin that does not apply packaging on the ingested stream. In our example, we are using HLS with 1-second segments pushed to AWS Elemental MediaStore. We run a command line, which will monitor the changes on the HLS child playlist on the origin. For example:

```
$ while sleep 0.01; do curl https://container.mediastore.eu-west-
1.amazonaws.com/livehls/index_730.m3u8 && date +"%Y-%m-%d
%H:%M:%S,%3N"; done
```

returns the time when segment "index_73020180223T154954_02190.ts" is first referenced in the shell output:

```
#EXTM3U
[...]
index_73020180223T154954_02190.ts
2018-02-23 15:49:55,515
```

We then download the segment "index_73020180223T154954_02190.ts" to verify the timecode it carries, which is 16:49:53:37 (UTC+1). We focus on the three last groups in the timecode (minutes, seconds and hundreds of seconds), as the hours can be different between your encoder and your workstation, depending on their timezone settings. In our case the minutes are identical, so we focus on the last two groups in the timecode. The difference between the current date and the segment timecode (above) is 55.51 - 53.37 = 2.14 seconds. By taking out the encoding latency and capture latency, we can determine the ingest latency. The formula is as follows:

```
(current date – segment timecode) – (capture latency + encoding latency) =
ingest latency
```

In this case, using AWS Elemental MediaStore, the ingest latency is .27 seconds.

MEASURING REPACKAGING LATENCY

In the above scenario with MediaStore, there was no repackaging. If you are doing any just-in-time packaging, as you can with AWS Elemental MediaPackage or AWS Elemental Delta, you'll want to measure your repackaging latency. You can take the same command line shell approach you took to measure ingest and apply it to repackaging:

```
(current date – segment timecode) – (capture latency + encoding latency +
ingest latency) = repackaging latency
```

Using AWS Elemental MediaPackage, the repackaging latency is .34 seconds. Using AWS Elemental Delta, it is .41 seconds.

MEASURING DELIVERY LATENCY

To measure the latency introduced by the content delivery network, you would use the same approach as for ingest and repackaging. If the origin does passthrough streaming, the formula would be as follows:

```
(current date – segment timecode) – (capture latency + encoding latency +
ingest latency) = delivery latency
```

If the origin server does repackaging, the formula would take that into account:

```
(current date – segment timecode) – (capture latency + encoding latency +
ingest latency + repackaging latency) = repackaging latency
```

Using AWS Elemental MediaStore and Amazon CloudFront delivery, running the same command line reveals a latency of .17 seconds.

MEASURING CLIENT LATENCY

Beyond encoding, server, and network latency, any remaining latency is caused at the client level. Client latency is the result of two factors: last mile latency and player latency. Last mile latency is the transfer time from CDN edge to the client, which is dependent on network bandwidth. You can calculate last mile latency by averaging the transfer time of 20 segments. To counteract last mile latency, you want to have a segment loading duration that is less than the segment length. If the loading duration is longer than the segment length, you will encounter rebuffering. Ideally, you want the segment loading duration to be less than 25% of the segment length; i.e., if you have 1-second segments, you want to keep the segment loading duration at .25 seconds or less.

Player latency is a consequence of the player's buffering policy (the player might require a certain number of segments to be buffered, or a specific minimum duration of content) and playhead positioning strategy. For example, a player can build a 10-segment buffer but start the playback at 3 segments from the end of the buffer, thus reducing the latency compared to the live edge time to only the length of 3 segments. Player latency varies widely from player to player, and later on we'll look at a few open source and commercial players, and how they can impact your overall latency.

So, you can determine the client latency using the following formula:

```
end-to-end latency – (capture latency + encoding latency + ingest latency +
repackaging latency + delivery latency) = client latency
```

EVALUATING TOTAL LATENCY

Now that we've determined the latency introduced by each component of the workflow, we have a sense of how much each component is responsible for. Using 1-second HLS segments produced with AWS Elemental Live and AWS Elemental MediaStore and delivered via Amazon CloudFront to a standard hls.js 0.8.9 player, we end up with the values shown in **Figure 5**. As you can see, the encoder and the player are responsible for the vast majority of latency—in this case, encoding introduces 1.48 seconds (almost 30%) of the latency, while the player is responsible for 2.55 seconds (almost 50%).

Latency Type	Seconds	Impact
Capture	0.39	7.58%
Encoding	1.48	**28.79%**
Ingest	0.27	5.25%
Repackaging	*N/A*	*N/A*
Delivery	0.17	3.33%
Last Mile	0.28	5.44%
Player	2.55	**49.61%**
End-to-end Latency	**5.14**	**100%**

Figure 5 – A breakdown of the latency added at each step of the workflow

More information about measuring latency can be found at go2sm.com/awslatency1. Next, we'll look at ways to optimize the various steps in the workflow, with the knowledge that we'll see the greatest improvements by optimizing the encoder and the player steps.

OPTIMIZING THE ENCODER

As mentioned earlier, the Low Latency Mode input parameter can be selected, but it may result in dropped audio packets, and the Buffer Size parameter can be reduced to the minimum at the input stage, but it may result in dropped frames. Beyond that, there are several video encoding parameters that can be adjusted to improve latency:

Lookahead: Setting it to low will improve latency, but since it reduces output quality for demanding scenes, it works well mainly if there are no or few scene changes.

GOP (group of pictures) parameters: We recommend using closed GOPs of 1-second duration that can be later on repackaged in 2-second segments if needed. Small GOPs without B Frames generally reduce video quality.

B Frames: The more B Frames used in a GOP, the higher the probability that encoding latency will be increased by a few frames for each B Frame added, as the encoding engine will look ahead to the next P Frames to construct the B frame. Using zero B Frames is possible to counteract this latency impact, but it will require raising the encoding bitrate to keep the same quality level as when using B Frames.

Temporal Adaptive Quantization (AQ): Turning it off will reduce latency by a few frames. The other Adaptation Quantization options don't impact latency.

Encoder Buffer Size: The default value is twice the video bitrate in bits, which generates a 2-second delay on the decoder. If set to 1x bitrate, it generates a 1-second delay and slightly impacts the video quality. For a very aggressive low latency target, the buffer size could be set to half of the bitrate, which results in half a GOP (1/2 second) of delay. Of course, the video quality will be affected; under 1/2x bitrate, the impact on video quality is more severe.

Video Preprocessors: If a Deinterlacer processor is required, the Low Latency interpolation algorithm should be selected.

Additionally, you can set the bottom rungs of your encoding ladder to very low bitrates for delivery to mobile devices under difficult network conditions. Players that have difficulty loading segments will still be able to receive the content without being impacted too negatively by the short segments.

OPTIMIZING THE PACKAGER

The most important consideration in the packaging step is the segment length. If you use a 1-second segment length, you will end up with roughly 5 seconds of end-to-end latency, assuming you optimize all the other steps. If you use a 2-second segment length, you will end up with between 7 and 10 seconds end-to-end latency. But beware that a 1-second segment length may create some problems on the player side, because the buffer will be smaller, and the playback session could suffer if the player doesn't have the right mechanism to cope with the smaller segment length. It's also important to note that shorter segment lengths can put pressure on the origin server. For example, if you have a 1-second segment length, the player will send 10 times as many requests to the origin server.

If you don't need latency below 7 seconds, use 2-second segments. If you use 2-second segments, it's also helpful to raise the GOP length from 1 to 2 seconds, so that you can increase the encoding quality at constant bitrate. You may also want to use 2-second segments when ingesting on your origin (if using HLS as an ingest format), to reduce stress on the origin storage and packaging computation.

The Index Duration—which is the depth of the DVR window that you are exposing in your playlist or manifest—also impacts latency. By default, some players buffer an amount of video, so the more you're exposing in the DVR window, the more the player will load at the start of the session. Depending on your player, you could sometimes end up with 40 seconds of startup buffer by default, if your DVR window is one hour.

Finally, if you are using AWS Elemental Live, you will need to lower the HLS/DASH Retry Interval publishing, because network transfer errors will need to be corrected faster. You should set the HLS/DASH Retry Interval to half of the segment length—for instance, 1 second for a 2-second segment, 0 seconds for a 1-second segment.

OPTIMIZING THE CDN

The CDN step of the workflow offers the fewest opportunities for optimization to decrease latency, but there are things you can do. Some CDNs introduce a buffer that can be disabled (this is not the case with Amazon CloudFront). Beyond that, you should make sure your CDN can serve gzipped HLS playlists or DASH manifests when long DVR windows are being used.

As players in low latency mode are generally aggressive in their requests compared to live edge time, there's a high chance that they will request segments in the future, resulting in 404s at the edge. Each CDN has a unique default TTL value for caching these 404s, and generally this value isn't friendly with low latency streams, so you need to adjust it. For an Amazon CloudFront distribution, you can set it to 1 second in the 'Error Pages' section of the configuration panel.
You also need to make sure that the playlist/manifest TTL (time-to-live) is shorter than or equal to your HLS segmentation interval or DASH manifest update interval.

Unrelated to low latency, but still important for your workflow: You need to whitelist the 'Origin' incoming header so that your CDN forwards it to the origin, as it is the key trigger for all downstream CORS policies returned by your origin. For Amazon CloudFront, you can set this in the 'Behaviors' section of the configuration panel.

More information on optimizing encoding, packaging, and CDN parameters can be found at go2sm.com/awslatency2.

OPTIMIZING THE PLAYER

Since the player can be responsible for up to 50% of the total end-to-end latency, you'll want to adjust the parameters in your target player to achieve the lowest latency it will allow. The default parameters in most players are set to favor uninterrupted playback, which means they prioritize buffer length over a lower stream latency. Here are some of the settings you can change to decrease the latency caused by the player.

INITIAL BUFFER SIZE

Most players default to buffer a specific number of segments, seconds, or megabytes (MB) before playback begins. If you are delivering 1- or 2-second segments and the player doesn't buffer more than three segments, you will likely end up with less than 10 seconds latency caused by the player. If, however, the player is designed to buffer a specific amount of time when a long DVR window is specified in the HLS playlist or DASH manifest, you might end up with 30 or 40 seconds of latency, even if your segments are only 1-second long. In most cases, you can set the playback buffer to between 2 and 4 seconds to find a reasonable tradeoff between latency and playback stability.

INITIAL PLAYHEAD POSITIONING

Most players rely only on the initial buffer size to position the playhead at startup, but some players use a complementary "initial delay" parameter to define the playhead position compared to the live edge time, expressed in seconds or in number of segments. In this case, a short initial buffer will be overridden by a longer initial delay.

LIVE EDGE-TIME STICKINESS

Even if your player starts playback with the expected delay compared to the live edge time, in the event of rebuffering it will likely resume playback at the last known timecode before the rebuffering. Some players allow you to reload the HLS playlist or DASH manifest after a dry buffer effect (when the audio or video track goes to 0 seconds and gets stuck), seek forward in time, or slightly accelerate the playback rate to stay close to the live edge time.

RESILIENCE TO SEGMENTS UNAVAILABILITY

If a particular segment is unavailable or available with some delay, the player will retry to load the segment a number of times and then stall playback if the segment is still not available. You may be able to adjust the player to increase the number of retry attempts, switch to lower bitrates, or skip a missing slice in the timeline.

Figure 6 shows how adjusting the settings of two popular open source players—dash.js and Shaka Player—can result in significant gains in latency. You can find more details for optimizing these players, as well as hls.js, Exoplayer, and several commercial players, at go2sm.com/awslatency3.

- **dash.js**
  - player.setLiveDelayFragmentCount (1) [default: 4]
  - player.setFragmentLoaderRetryInterval (300) [default: 1000ms]
  - player.setStableBufferTime (2) [default: 12s]
  - player.setBufferToKeep (2) [default: 20s]
  - player.setBufferTimeAtTopQuality (2) [default: 30s]

Player	3x1s DASH	3600x1s DASH
dash.js 2.6.5	6.50s	**10.38s**
dash.js 2.6.5 (optimized)	6.47s	**7.10s**

- **Shaka player**
  - streaming.bufferingGoal (2) [default: 10s])

Player	3x1s DASH	3600x1s DASH
Shaka 2.3.0	**6.94s**	6.44s
Shaka 2.3.0 (optimized)	**5.54s**	6.01s

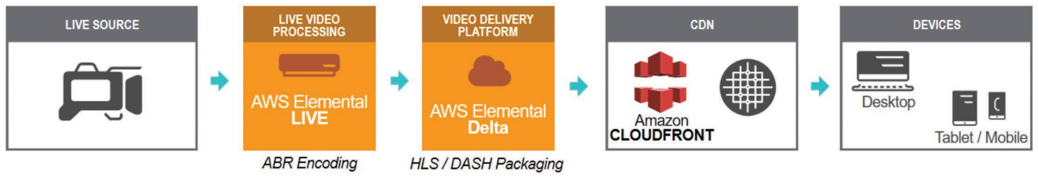
Figure 6 – Optimizing two open source players to decrease latency. (Note that dash.js 2.6.5 is an older version of the player, and much has been done to improve latency since v2.7. We use 2.6.5 here simply to demonstrate that players' default parameters are not always designed for low latency.)

DECREASING LATENCY WITH AWS ELEMENTAL WORKFLOWS

Now let's look at four workflow examples—one deployed on-premises and three hybrid scenarios with encoding or contribution on-premises and the rest of the workflow in the cloud—using AWS Elemental to decrease latency. These examples can help you visualize what to expect from your existing equipment and services, and how to combine them with AWS Elemental services to achieve the desired latency for your requirements.

ON-PREMISES

In this scenario, we are using AWS Elemental Live for encoding and AWS Elemental Delta for packaging and origin. As **Figure 7** shows, we are able to achieve latencies ranging from 5.49 seconds to 11.65 seconds. The gray boxes indicate scenarios where a player is incompatible with a protocol—dash.js doesn't play HLS, and hls.js and Safari mobile don't play DASH.

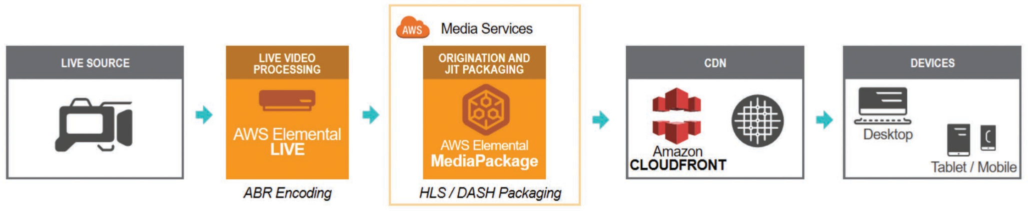


Player	3x1s HLS (TS)	3x2s HLS (TS)	3x1s DASH	3x2s DASH
hls.js 0.8.7	6.86s	8.44s		
dash.js 2.6.5			5.94s	8.23s
Safari mobile (iOS 11.2.2)	6.04s	11.65s		
Exoplayer 2.6.0 (Android 6.0.1)	6.14s	10.19s	5.49s	7.14s

Figure 7 – Latency results for an on-premises workflow using AWS Elemental Live for encoding and AWS Elemental Delta for packaging and origin

HYBRID SCENARIO 1

The first hybrid scenario uses AWS Elemental Live for on-premises encoding, AWS Elemental MediaPackage for repackaging, and Amazon CloudFront for delivery. As you can see in **Figure 8**, the repackaging step introduces some latency, so our numbers range from 5.69 seconds to 10.59 seconds—not a significant difference from the on-premises scenario. We recommend producing 2-second segments in HLS and DASH; you'll need to tune the player if you produce 1-second segments.

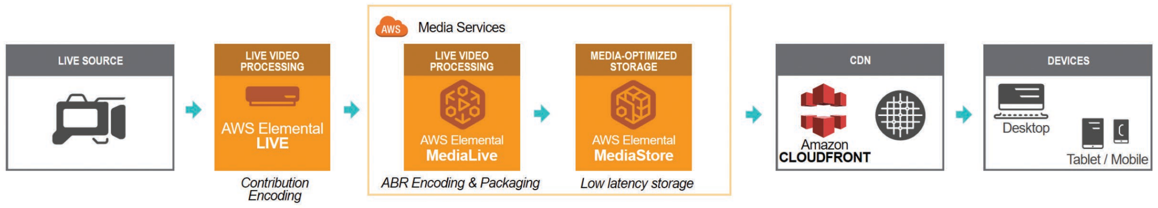


PLAYER	3x1s HLS (TS)	3x2s HLS (TS)	3x1s DASH	3x2s DASH
hls.js 0.8.7	6.39s	9.35s		
dash.js 2.6.5			5.69s	7.54s
Safari mobile (iOS 11.2.2)	6.64s	10.64s		
Exoplayer 2.6.0 (Android 6.0.1)	6.95s	10.59s	7.19s	8.11s

Figure 8 – Latency results for a hybrid workflow using AWS Elemental Live for encoding, AWS Elemental MediaPackage for repackaging, and Amazon CloudFront for delivery

HYBRID SCENARIO 2

In this scenario, contribution is on-premises, but transcoding is done in the cloud with AWS Elemental Live and then the HLS streams are pushed to AWS Elemental MediaStore origin. Here we were able to achieve latency of less than 7 seconds when using 1-second segments and a Buffer Size of half the bitrate; 2-second segments resulted in longer latency (see **Figure 9**).

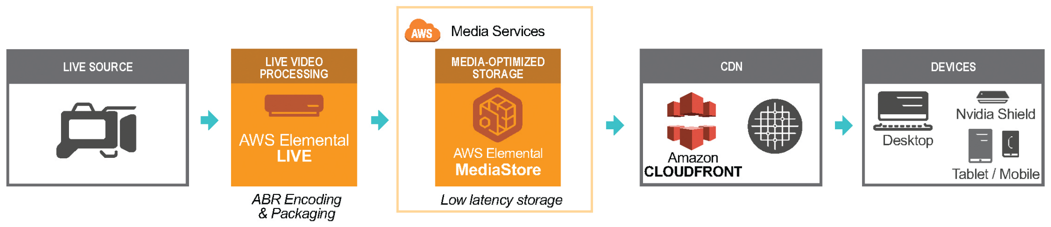


PLAYER	3x1s HLS (TS)	3x2s HLS (TS)
hls.js 0.8.7	6.64s	9.85s
Safari mobile (iOS 11.2.2)	6.68s	12.56s
Exoplayer 2.6.0 (Android 6.0.1)	8.86s	11.94s

Figure 9 – By doing contribution encoding on-premises and using the cloud for both transcoding and origin server, we were able to achieve latencies of less than 7 seconds when using 1-second segments

HYBRID SCENARIO 3

This scenario includes two additional players, Shaka Player 2.3.0 (standard) and Shaka Player 2.3.0 (optimized). We also upgraded the dash.js player to 2.7.0, which offers a new low-latency mode by leveraging the fetch API. We also added a new format, fragmented MP4 (fMP4) for HLS, also known as CMAF (Common Media Application Format), which replaces the Transport Stream (TS) segments with MP4 segments. Note that not all players can take advantage of the new format. Finally, we introduced 1-hour DVR windows (3600sx1s) to see how players behave with long index duration. As shown in **Figure 10**, latencies are now down below 10 seconds, and in some cases less than 5 seconds.



PLAYER	3x1s HLS	3600x1s HLS	3x1s DASH	3600x1s DASH
hls.js 0.8.7	5.32s (TS) 5.34s (fMP4)	6.30s (TS)		
dash.js 2.7.0			4.90s	6.20s
Safari mobile (iOS 11.2.2)	5.64s (TS) 7.50s (fMP4)	5.34s (TS) 9.47s (fMP4)		
Exoplayer 2.6.0 (Android 6.0.1)	5.31s (TS) 6.25s (fMP4)	7.49s (TS) 7.22s (fMP4)	6.40s	7.06s
Shaka 2.3.0 [optimized]		6.80s (TS)	5.54s	6.01s

Figure 10 – By using an optimized Shaka Player 2.3.0, an upgraded dash.js player (2.7.0), and adding fragmented MP4 (fMP4), we are able to achieve much lower latencies than in the previous scenarios

More details about each of these scenarios can be found at go2sm.com/awslatency4.

SO WHAT ARE YOU WAITING FOR?

The days of viewers being satisfied with latency in the 30-second range are over. Today, viewers expect live events on OTT to compete with broadcast, which means latency in the 10-second range, and latency in the 4- to 5-second range is now achievable with the right workflow and player. By leveraging HLS and DASH, and fine-tuning the parameters at each step in the workflow, you can compete with (and in some cases beat) broadcast latency. That means more satisfied consumers, which translates into increased revenue for your OTT service.