Streaming Techniques

Written by [Febogamedev]

In the following sections, we will examine the techniques that support streaming.

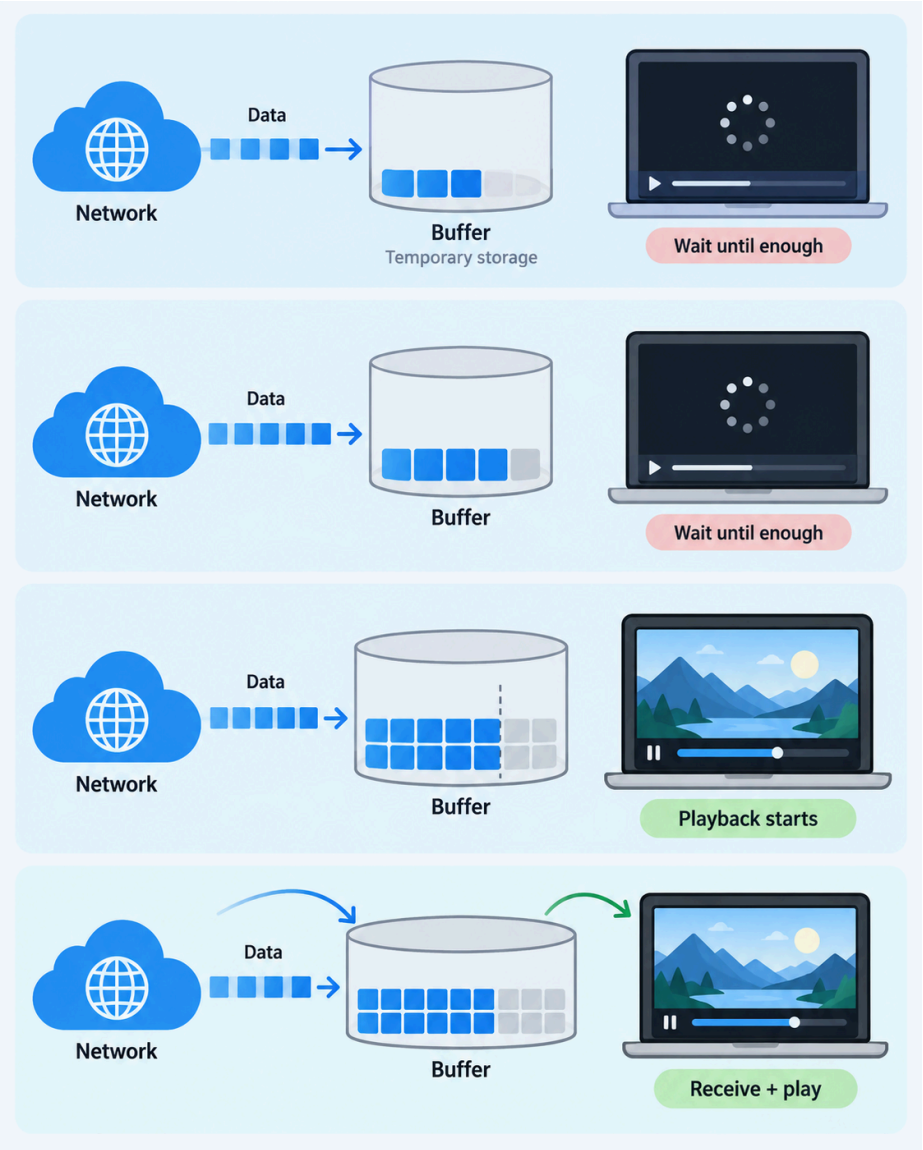
Buffering

Buffering is the process through which a device receives and temporarily stores part of the multimedia content before it is played (DASH Industry Forum, n.d.-b).
Its aim is to keep enough data already available on the device so that playback can continue even when network speed temporarily varies.

When a user starts streaming content, the process for a player can be simplified as follows:

- It progressively receives data from the network.
- It temporarily stores it in the buffer.
- It waits until enough content is available.
- It begins playback.
- It continues receiving new data while the data already present is played.

The buffer is therefore continually filled by the network and consumed by the player.



For example, if the player already has a few seconds of video available and the connection slows down temporarily, it can continue playing what is already in the buffer.
However, if the buffer empties before new data arrives, playback must stop. This is called **rebuffering** and is normally perceived by the user as a pause accompanied by a loading indicator (DASH Industry Forum, n.d.-b).

Example
Imagine watching a film through streaming.
Before beginning playback, the player accumulates five seconds of content:
Buffer: [1][2][3][4][5].

When the first second is played:
Playing 1 → Buffer: [2][3][4][5].

Meanwhile, another second of video arrives:
6 arrives → Buffer: [2][3][4][5][6].

Now suppose that the connection cannot supply new data for three seconds.
The player can still continue:
Playing 2 → [3][4][5][6].
Playing 3 → [4][5][6].
Playing 4 → [5][6].

Nothing visible happens for the user because enough data was already available.
However, if the problem continues:
Playing 5 → [6].
Playing 6 → Empty buffer.

At this point, the player must stop playback and wait for new data to arrive.

The buffer's ability to maintain continuous playback depends on several factors:

- Connection speed: If the average rate at which data arrives is lower than the bitrate needed to play the content, the buffer tends to empty progressively.
- Buffer size: A larger buffer can absorb longer network fluctuations, but may increase the time needed before playback begins and, particularly with live content, latency.
- Network congestion: If many devices share the same connection, the bandwidth available for streaming may decrease.
- Network stability: A connection with large variations in transfer speed makes it harder to keep the amount of available data constant.
- Content bitrate. High-bitrate video requires more data for each second of playback.



The fundamental relationship is therefore:

- Data received > data consumed → the buffer grows.
- Data received ≈ data consumed → the buffer remains stable.

- Data received < data consumed → the buffer empties.

During a streaming transmission, problems may arise that concern not only how much data is available, but also when that data arrives.
Two particularly important concepts are **latency** and **jitter**.
In live streaming, latency can be understood as the time between what happens in front of the camera and the moment the viewer sees it (Amazon Web Services, n.d.-h).

Example
For example, when watching a live event, there may be a five-second difference between what is happening at that moment and its display.
Real event → 5 seconds → Display.

Overall latency can be introduced by numerous stages during streaming:
Capture → Encoding → Ingest → Network → Server → Distribution → Buffer → Decoding → Playback.

Each of these stages can introduce a small amount of delay.
Latency therefore depends on factors such as:
- Content encoding.
- Distance and path through the network.
- Processing performed by servers.
- Segment size.
- The buffer used by the player.
- Decoding performed by the device.

However, it is important to remember that latency does not have the same importance in every service.
- When watching an on-demand film, a few seconds of initial delay are generally of little consequence.
- In a videoconference, a call or other interactive applications, however, even a few hundred milliseconds can make communication less natural.

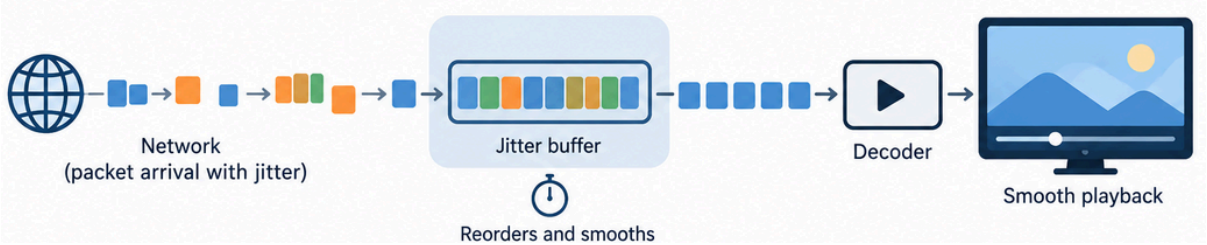
Jitter is the variation in the delay with which packets reach their destination (Schulzrinne et al., 2003).
- Ideally, if they are sent at regular intervals, they should arrive with similar timing: 10 ms → 10 ms → 10 ms → 10 ms.
- On a real network, packets might instead arrive like this: 10 ms → 35 ms → 8 ms → 27 ms.

The problem is not necessarily that packets fail to arrive, but that they may arrive too early, too late or at irregular intervals.
This is particularly problematic for real-time audio and video because each packet must be available at the right moment to be played.

High jitter can cause:
- Interruptions or stuttering during playback.

- Unusable packets because they arrive too late.
- Brief silences or audio distortion.
- Missing frames or frozen video.
- Audio/video synchronisation problems.

To compensate for these variations, a special type of buffer called a j**itter buffer** is used. This specialised buffer is mainly used in real-time communication (W3C, 2025).
Unlike a normal streaming buffer, which accumulates a certain amount of content to protect playback from temporary data shortages, a jitter buffer primarily seeks to regularise the timing with which packets are delivered to the decoder.



Example
Imagine a video call in which an audio packet is sent every 20 milliseconds.
Ideally:
A → 0 ms.
B → 20 ms.
C → 40 ms.
D → 60 ms.
Playback can therefore proceed regularly:
A → B → C → D.

However, the network might deliver them like this:
A → 0 ms.
B → 24 ms.
C → 67 ms.
D → 71 ms.

Packet C arrived much later than expected, while D arrived almost immediately afterwards.
If packets were played at exactly the moment they arrived, the timing would be irregular:
A —— >B ———————— >C->D.

The jitter buffer therefore introduces a small intentional delay.
Instead of immediately delivering each packet to the decoder, it waits briefly and tries to reconstruct a more regularly timed sequence:
A → B → C → D.
20 ms → 20 ms → 20 ms.

However, the jitter buffer cannot wait indefinitely.

If a packet arrives after the moment at which it should have been played, it may be considered too late and therefore no longer usable (W3C, 2025).
Here too, there is therefore a trade-off:

- Larger jitter buffer → greater ability to absorb irregularities, but higher latency.
- Smaller jitter buffer → lower latency, but greater sensitivity to jitter.

The fundamental distinction is therefore:

- The buffer mainly protects against temporary shortages of available data.
- The jitter buffer mainly protects against irregular packet arrival timing.

Example
An analogy can help distinguish a buffer from a jitter buffer.

Imagine a waiter who must continually serve dishes.
A normal buffer is equivalent to having several dishes already prepared.

If the cook slows down for a few minutes, the waiter can continue serving those available.

A jitter buffer, by contrast, is equivalent to receiving dishes from the kitchen at irregular intervals:
20 s → 40 s → 5 s → 15 s.
The waiter holds them briefly and tries to serve them more regularly:
20 s → 20 s → 20 s → 20 s.

The analogy is simplified, but highlights the main difference:

- In the first case, the concern is having a sufficient reserve.
- In the second, it is maintaining regular timing.

During playback, we can distinguish several phenomena related to buffer management.

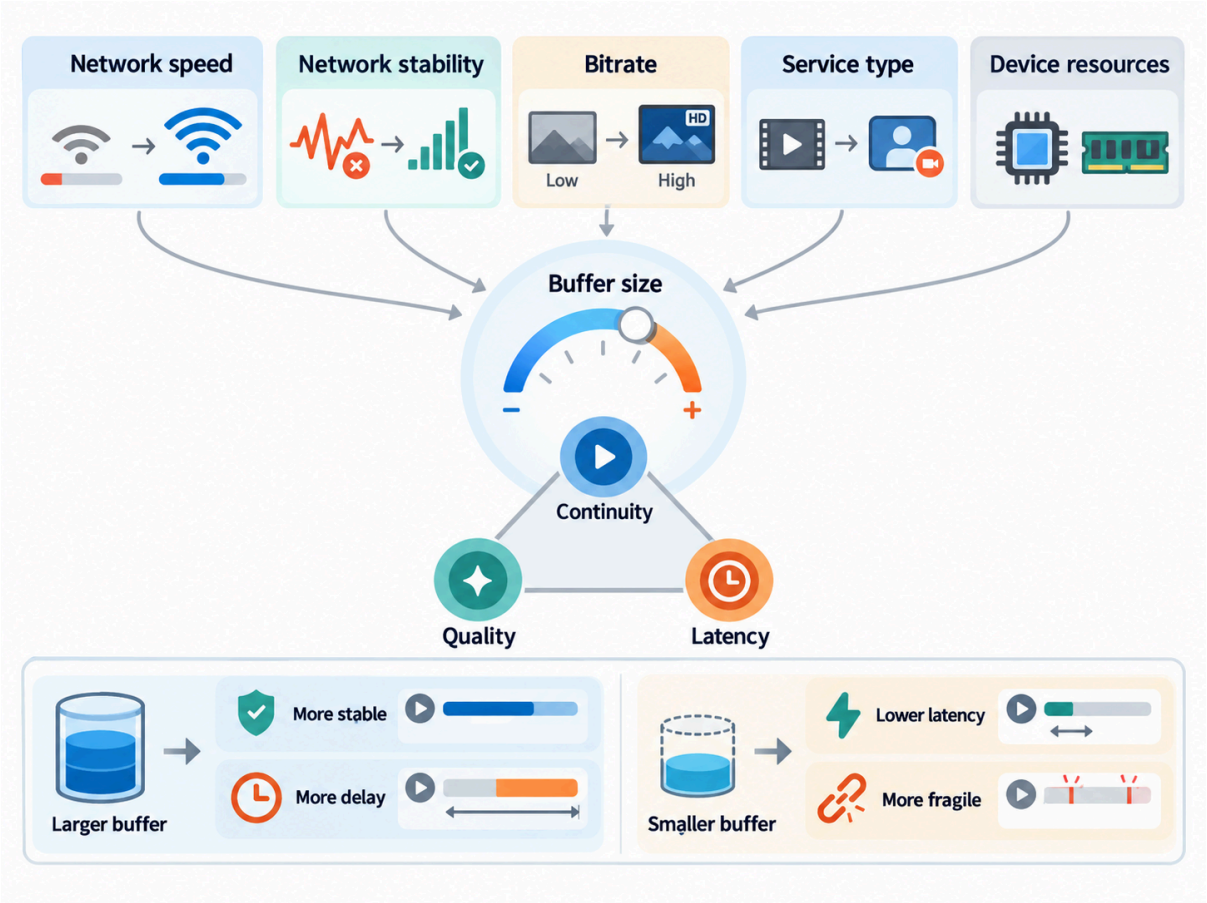
- **Initial buffering** is the initial loading performed before playback starts. The player waits until enough data is available to reduce the risk of an immediate interruption.
- **Rebuffering** occurs when the buffer empties during playback. The player is forced to stop and wait for new data to arrive.

Some players can dynamically adjust the amount of data they try to keep in the buffer according to network conditions (DASH Industry Forum, n.d.-b).
Buffer size is therefore an important design choice and depends on several factors:

- Network speed: a fast network generally allows the buffer to fill quickly.
- Network stability: an unstable network may require a larger reserve of data.
- Content bitrate: a high bitrate requires more data for each second of playback.
- Type of service: an on-demand film can tolerate a relatively large buffer; real-time communication instead requires much smaller buffers.
- Device resources: available memory and device characteristics can influence the player's strategy.

Designing buffering therefore involves finding a balance between three needs:
Continuous playback ↔ Quality ↔ Latency.

- Increasing the amount of accumulated content can make streaming more resistant to network fluctuations, but may increase the initial wait and, in live streaming, the delay relative to the event. Maximum buffer capacity alone does not automatically determine this delay: the content actually accumulated and the player's strategy are what matter (DASH Industry Forum, n.d.-b).
- Reducing accumulated content can decrease delay, but makes playback more vulnerable to connection irregularities.



For this reason, there is no universally ideal buffer size: it depends on the type of streaming and the experience we want to achieve.

Adaptive Bitrate Streaming

Modern systems try to avoid rebuffering by using **Adaptive Bitrate Streaming (*ABR*)** techniques.

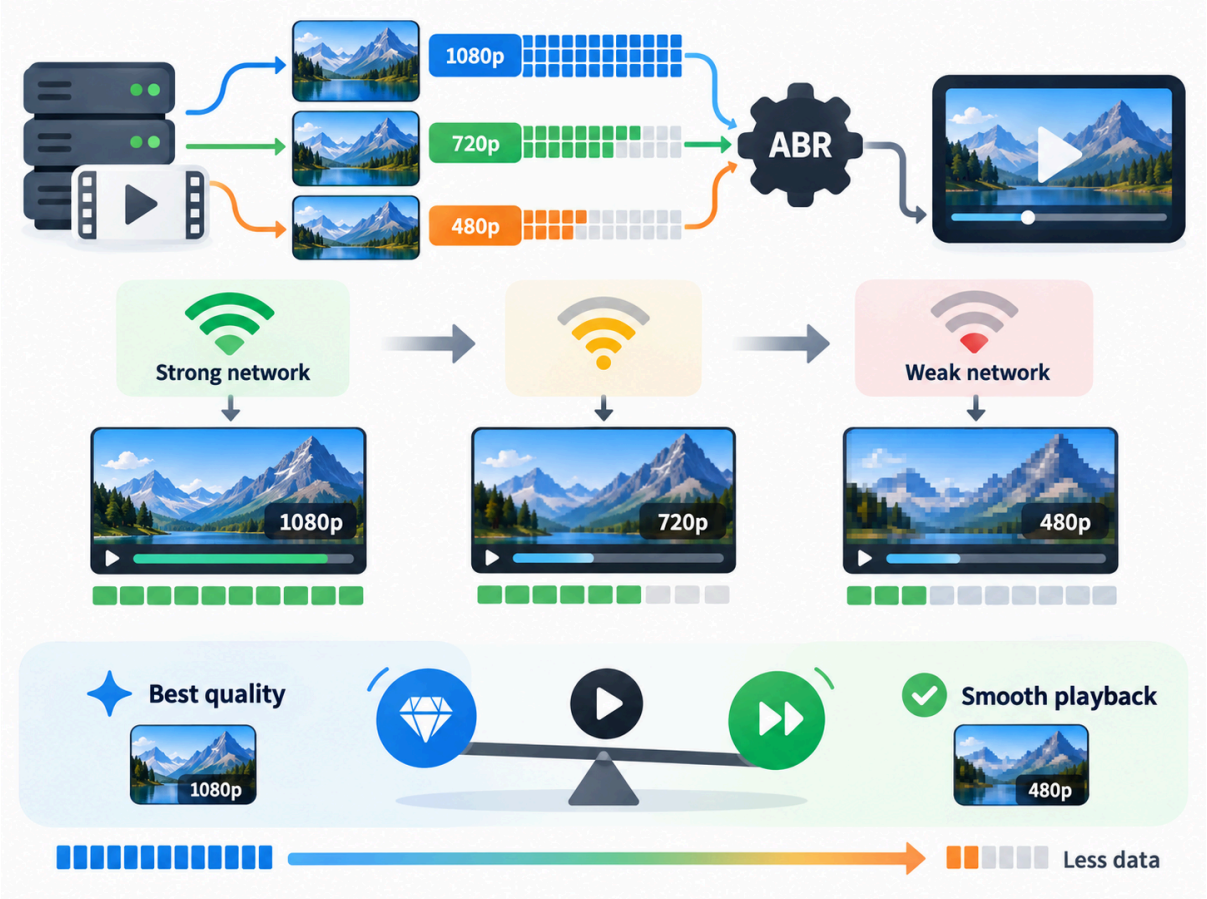
ABR is a technique that allows the player to automatically adjust content quality during playback according to the available conditions (DASH Industry Forum, n.d.-a).
If the network deteriorates, the player can request a lower-bitrate version of the content, reducing the amount of data needed for each second of video.
For example:

1080p → insufficient network capacity → 720p → 480p.
In this way, the player temporarily sacrifices some image quality to maintain continuous playback.

The aim is to continually balance two needs:
Highest possible quality ↔ Continuous playback.

- If the network can transfer large amounts of data, the player can use a version of the content with higher quality and bitrate.
- If network capacity decreases, it can temporarily select a version that requires less data, reducing the risk of the buffer emptying.

ABR is one of the fundamental features of streaming systems such as *HLS* and *DASH*.



But how does *ABR* actually work?
Before distribution, the same content is encoded into different versions, generally called **renditions** or **variants**.
These versions correspond to a particular resolution and bitrate.
For example:

- 1080p → 6 Mbit/s.
- 720p → 3 Mbit/s.
- 480p → 1 Mbit/s.

The set of available versions is often called a **bitrate ladder** (Amazon Web Services, n.d.-g).

Further exploration
Resolution and bitrate are related, but they are not the same thing.
Two 1080p versions can use different bitrates.
The bitrate required also depends on factors such as:

- The codec used.
- Frame rate.
- Image complexity.
- Type of content.
- Desired quality.

Each version is then divided into temporally corresponding segments.
For example:

- 1080p: [1][2][3][4][5]
- 720p: [1][2][3][4][5]
- 480p: [1][2][3][4][5]

The segments represent the same temporal portions of the video, but with different characteristics. This allows the player to switch versions during playback (DASH Industry Forum, 2018).

The player needs to know which versions of the content are available and where to find their segments.
This information is provided through a **manifest**.

- In *HLS*, the information needed for playback is organised into files called **playlists**, generally in .m3u8 format. A master playlist describes the different available versions of the content, while the playlists associated with each version identify the media segments to download and play (Pantos & May, 2017).
- *DASH* instead uses a manifest file called an ***MPD*** **(Media Presentation Description)**. *The MPD* describes the different available representations of the content, providing information such as bitrate, resolution, codec and how to retrieve their segments (DASH Industry Forum, 2018).

But how does this process work during live streaming?
During a live transmission, new versions of the stream and new segments are continually generated.
For example:

- Live 1080p → [101][102][103] → …
- Live 720p → [101][102][103] → …
- Live 480p → [101][102][103] → ...

The manifest is updated as the transmission proceeds, and the player continues requesting newly available segments (Pantos & May, 2017).

ABR allows the player to automatically select the most suitable version based on the conditions and criteria considered.

During playback, the player continually observes the available conditions and decides which version to request for subsequent segments.
The process can be simplified as follows:
Measurement → Quality selection → Segment download → Playback → New measurement.



Example
Suppose the video is available in three versions:

- 1080p → 6 Mbit/s.
- 720p → 3 Mbit/s.
- 480p → 1 Mbit/s.

Initially, the player might choose: Segment 1 → 1080p.
The network deteriorates: Segment 2 → 720p.
The network deteriorates further: Segment 3 → 480p.
It subsequently improves: Segment 4 → 720p.
And finally: Segment 5 → 1080p.

Playback therefore continues to represent the same content, but at different resolutions:
[1080p] → [720p] → [480p] → [720p] → [1080p]
This gives the user a smooth viewing experience without having to select a new version manually.

This is possible because the different variants represent the same time intervals of the content.

To decide which version of the content to request, the player uses an *ABR* algorithm.
An *ABR* algorithm can consider several factors simultaneously:

- The speed at which previous segments were downloaded.
- The amount of content still in the buffer.
- The bitrates of the available versions.
- Recent connection stability.
- Device characteristics and capabilities.
- Screen resolution or player size.

For example, even if the network appears fast enough for 1080p, the player might temporarily choose 720p if the buffer is almost empty (DASH Industry Forum, n.d.-a).
The decision might therefore be:

- Fast network + well-filled buffer → higher quality.
- Uncertain network + low buffer → lower quality.

This allows the system to avoid overly aggressive changes that could quickly lead to rebuffering.
From this, we can deduce that *ABR* and buffering work together.

- The buffer provides a temporary reserve of content.
- *ABR* instead tries to select a quality level that allows that reserve to be maintained (DASH Industry Forum, n.d.-a).

ABR offers several advantages:

- More continuous playback: it reduces the risk of rebuffering when network conditions change.
- Adaptive quality: it allows high quality when conditions permit and lower quality when necessary.
- Adaptation during playback: content does not need to be interrupted to switch versions.
- Efficient use of available bandwidth: the player can avoid requesting unnecessarily high bitrates when conditions or the device make them inappropriate.
- Support for live and on-demand streaming: the same principle can be used for both modes.

However, it also has some limitations:

- Variable quality: an unstable network can cause visible quality changes.
- Greater processing complexity: the provider must produce and manage different versions of the same content.
- Greater storage requirements: multiple renditions must be stored for on-demand content.
- Greater infrastructure complexity: encoders, packagers, manifests, *CDNs* and players must correctly coordinate the different variants.
- Higher encoding costs: each piece of content must be encoded into multiple versions.

Content Delivery Network

A *CDN* (Content Delivery Network) is an infrastructure consisting of servers distributed across numerous network locations to deliver content to users more efficiently (Netflix, n.d.).
It can be used to distribute:

- Video and audio.
- Images.
- Files.
- Web pages and resources.
- Segments and manifests used in streaming.

In streaming, a *CDN* prevents all users from having to retrieve content directly from the origin server.

- Without a *CDN*: Origin Server → User
- With a *CDN*: Origin Server → *CDN* → User

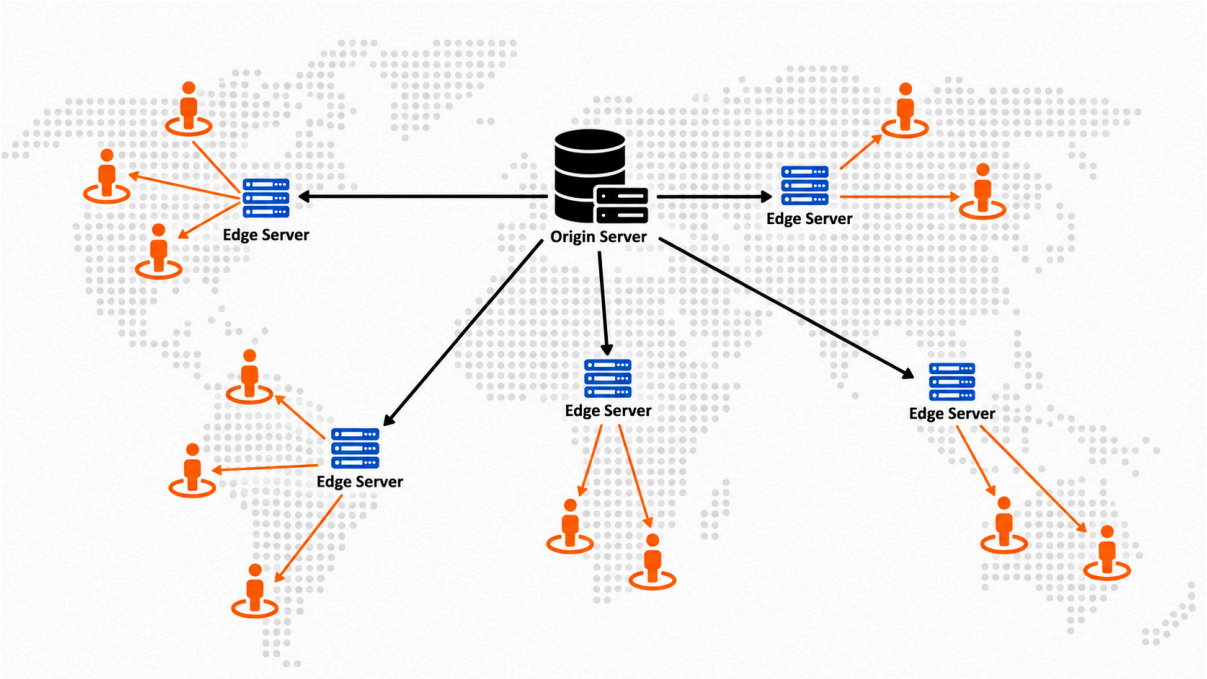
If millions of users simultaneously requested the same segments directly from the origin server, it would have to handle an enormous amount of traffic.
The *CDN* instead distributes requests among numerous servers located at different points in the network.
The aim is to:

- Reduce latency.
- Reduce the load on the origin server.
- Increase the ability to serve many users simultaneously.
- Reduce the risk of buffering caused by distribution.
- Make global content delivery more efficient.

The original content is generally made available by an origin server.
The CDN can retrieve content from the origin server on request or distribute it in advance to its Points of Presence (PoPs), depending on the strategy adopted.
PoPs contain **edge servers**: the servers that can deliver content to users.

- *PoP*: the physical/logical point of the *CDN* network in a particular geographical area. It may be a data centre or network facility.
- Edge server: one of the servers within a *PoP* that actually receives user requests and delivers content.

The path can therefore be represented as follows:
Origin Server → *PoP* → Edge Server → User



When a user requests content, the *CDN* tries to direct them to a *PoP* suitable for serving that request.
This does not necessarily mean choosing the physically closest server.
Network characteristics, such as the available path and latency, are also considered.
Amazon CloudFront (the *CDN* service managed by Amazon Web Services), for example, describes this process as routing to the *PoP* that can best serve the request, generally the closest in terms of latency (Amazon Web Services, n.d.-c).

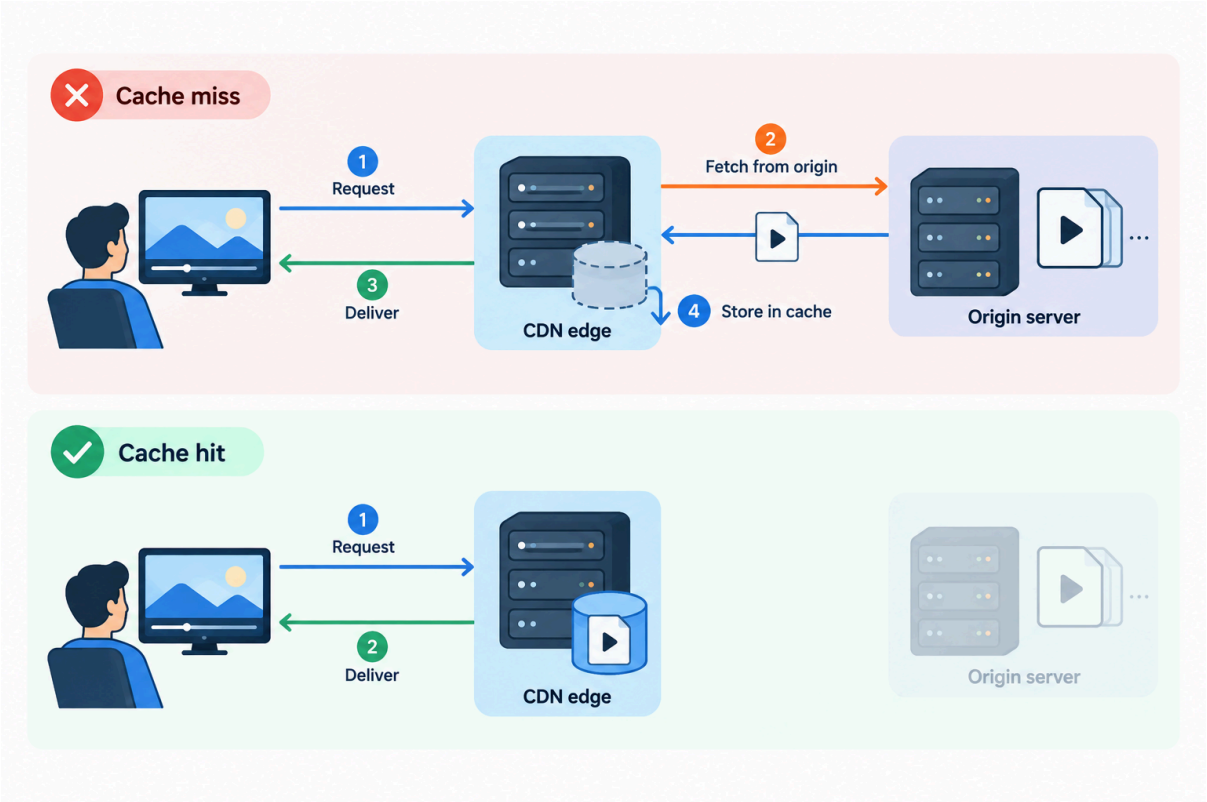
One of the fundamental mechanisms of *CDNs* is **edge caching**.
Edge servers can temporarily store copies of content requested by users.
When a request arrives, two situations may arise.

- Cache hit.
- Cache miss.

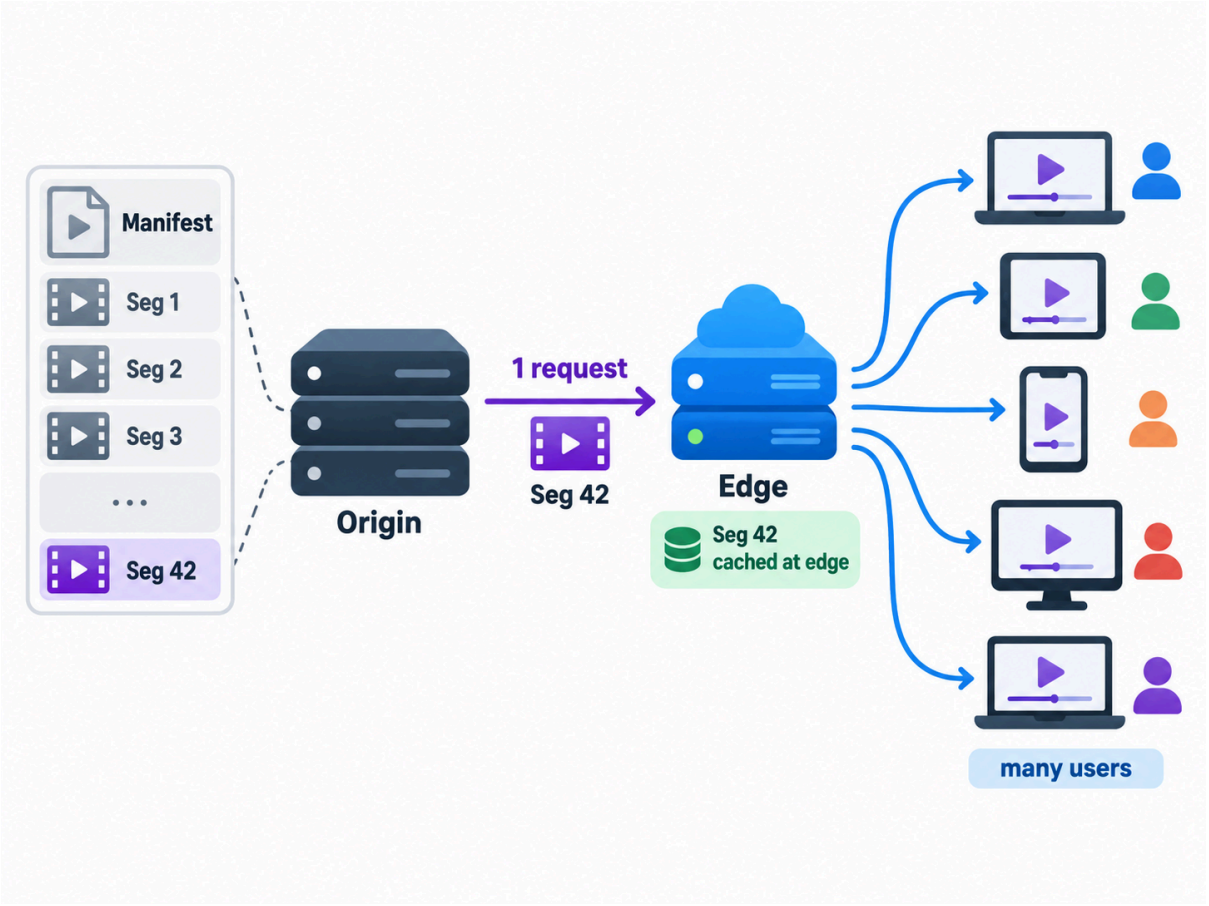
With a cache hit, the requested content is already in the edge server's cache. If the copy is still valid and reusable under HTTP rules, it can be served without contacting the origin again; in other cases, validation may be necessary (Fielding et al., 2022).
This reduces both latency for the user and traffic directed to the origin.

With a cache miss, the content is not in the cache.
The edge therefore retrieves the content from the origin or an intermediate cache layer and sends it to the user (Amazon Web Services, n.d.-c).
In many systems, it can also store it in its own cache so that a subsequent request for the same content can be served directly by the edge server (Amazon Web Services, n.d.-c).
The strategy can therefore be summarised as follows:

- First request: User → Edge → Origin.
- Subsequent requests: User → Edge.



The segmentation used by technologies such as *HLS* and *DASH* is particularly well suited to *CDN* distribution.
A video may consist of:
Manifest + Segment 1 + Segment 2 + Segment 3 + … + Segment 42 + …
Suppose thousands of users are watching the same video.
The segment “segment42.m4s” might initially be requested from the origin
and subsequently be delivered by the edge to many users.
The origin therefore does not have to send the same segment separately to every viewer.
This makes *CDNs* particularly suitable for large-scale video services.



Using a *CDN* offers several advantages.

- Lower latency: content can be served from network nodes better suited to the user instead of contacting the origin server every time.
- Reduced load on the origin: the same cached copy can satisfy numerous requests.
- Greater scalability: traffic is distributed among numerous servers instead of being concentrated on a single machine.
- Reduced network traffic: content already present at the edges does not need to be repeatedly retrieved from the origin.
- Greater resilience: a distributed network can use multiple nodes and paths to continue providing the service.
- Efficient support for *HLS* and *DASH*: manifests and segments can be distributed through the CDN's HTTP infrastructure.

However, *CDNs* do not automatically eliminate buffering and latency: they reduce some of their possible causes.
Final quality also depends on the user's connection, bitrate, player, *ABR*, buffer and overall network conditions.

There are different ways in which a *CDN* can obtain the content it distributes.
A useful, though simplified, distinction is between **pull caching** and **pre-positioning/push**.

In the pull model, content is retrieved from the origin when it is requested.
It works as follows:

User requests content → Edge checks the cache.

Depending on the result of the cache check, two cases may arise:

- Cache hit → deliver the content.
- Cache miss → retrieve content from the origin → deliver it → optionally store it.

This model has the advantage of storing mainly content that is actually requested at the edges.
The disadvantage is that the first request may require access to the origin.

In the push model, by contrast, some content can be distributed to edge nodes in advance, before a user requests it.
This approach can be useful when it is possible to predict which content will be in high demand.
The disadvantage is that network capacity and storage space are used even before the content is actually requested.

Real systems can combine both strategies.
An interesting example is Netflix Open Connect (Netflix's *CDN*): Netflix can preload its Open Connect Appliances (Netflix's edge servers) with content appropriate for a particular area and subsequently update them (Netflix, n.d.).

CDNs are also used in live streaming.
In this case, segments are generated continuously during the event and distributed as they become available.
Naturally, these segments cannot be preloaded before the event exists.

A traditional *CDN* is primarily concerned with content distribution and caching.
However, some modern infrastructures also allow processing to be performed directly at the network's edge nodes (edge servers).
This is referred to as **edge computing** (Amazon Web Services, n.d.-a).

- The traditional model is: User → Edge → Origin → Processing
- With edge computing, some operations can be performed directly near the user: User → Edge → Processing

This can be used, for example, to:

- Modify a response based on the request.
- Execute application logic.
- Check authentication and authorisation.
- Transform or optimise certain content.
- Manage routing and personalisation.
-

However, it is useful to keep the two concepts distinct:

- *CDN* → brings content closer to users.
- Edge computing → also brings processing closer to users.

Example

Imagine a user requesting protected video. In a traditional *CDN*, the edge might simply look for the content in the cache and, if necessary, retrieve it from the origin.
With edge computing, the edge can also execute application logic before returning the content:
User → Edge → authorisation check → content

For example, the edge can check:
- Whether the user's token is valid.
- Whether the subscription allows that content to be viewed.
- Whether the content is available in the user's country.
- Whether the request should be accepted or rejected.

The flow might be:
- User → Edge → Token check
- If the token is valid: Edge → Cache/CDN → Video
- If it is invalid: Edge → Access denied

The advantage is that the request is processed near the user, without having to query a central server every time (Amazon Web Services, n.d.-a).

Major commercial *CDN* providers include:
- Akamai.
- Cloudflare.
- Amazon CloudFront.
- Fastly.

Some companies have also developed *CDN* infrastructures specifically for their own services.
An important example is Netflix Open Connect, which uses cache servers called Open Connect Appliances (*OCAs*).
These servers can be placed at Internet exchange points or directly within *ISP* infrastructures, allowing Netflix content to be served very close to users (Netflix, n.d.).

Cloud computing

Cloud computing allows computing resources such as processing capacity, storage, databases and software services to be used over the network without having to directly own and manage all the necessary physical infrastructure (Mell & Grance, 2011).
In streaming, the cloud can be used for numerous operations:
- Content storage.
- Audio/video encoding and transcoding.
- Packaging for *HLS* and *DASH*.
- User management and authentication.
- Data processing.
- Analytics and recommendation systems.
- Managing and coordinating the streaming infrastructure.

The cloud therefore primarily represents the central processing and management component of many modern architectures, while final content delivery is often entrusted to *CDNs* and edge servers.

The general model can be represented as follows:
Cloud / Origin → *CDN* → Edge Server → User.

A cloud server is a computing resource available within a cloud infrastructure.
In its most traditional form, it may be a virtual machine running within a remote data centre.
Instead of purchasing and physically maintaining a server, an organisation can use resources provided by a cloud provider, such as:

- Amazon Web Services (*AWS*).
- Microsoft Azure.
- Google Cloud.

One of the main advantages is the ability to adjust available resources according to needs.
For example, a service might use 10 servers during a normal period and temporarily increase to 100 during a sharp rise in traffic.
This capability is called elasticity: the ability to dynamically increase or reduce resources according to load (Mell & Grance, 2011).
However, the modern cloud does not consist solely of virtual servers.
A platform can also use object storage (a storage architecture that manages data as independent “objects” rather than within a folder hierarchy), containers, managed databases, serverless functions (small, self-contained pieces of code executed in the cloud in response to specific events) and specialised services without directly administering the individual machines that run them.

Multimedia content requires large amounts of space.
For this reason, **cloud storage** services are often used to store large amounts of data within the provider's infrastructure.
One example is Amazon S3, an object storage service (Amazon Web Services, n.d.-f).

The original file is not used solely for storage: it can also subsequently be processed to produce the versions intended for streaming.

It is therefore important to distinguish between:

- Compute → performs processing.
- Storage → stores data.

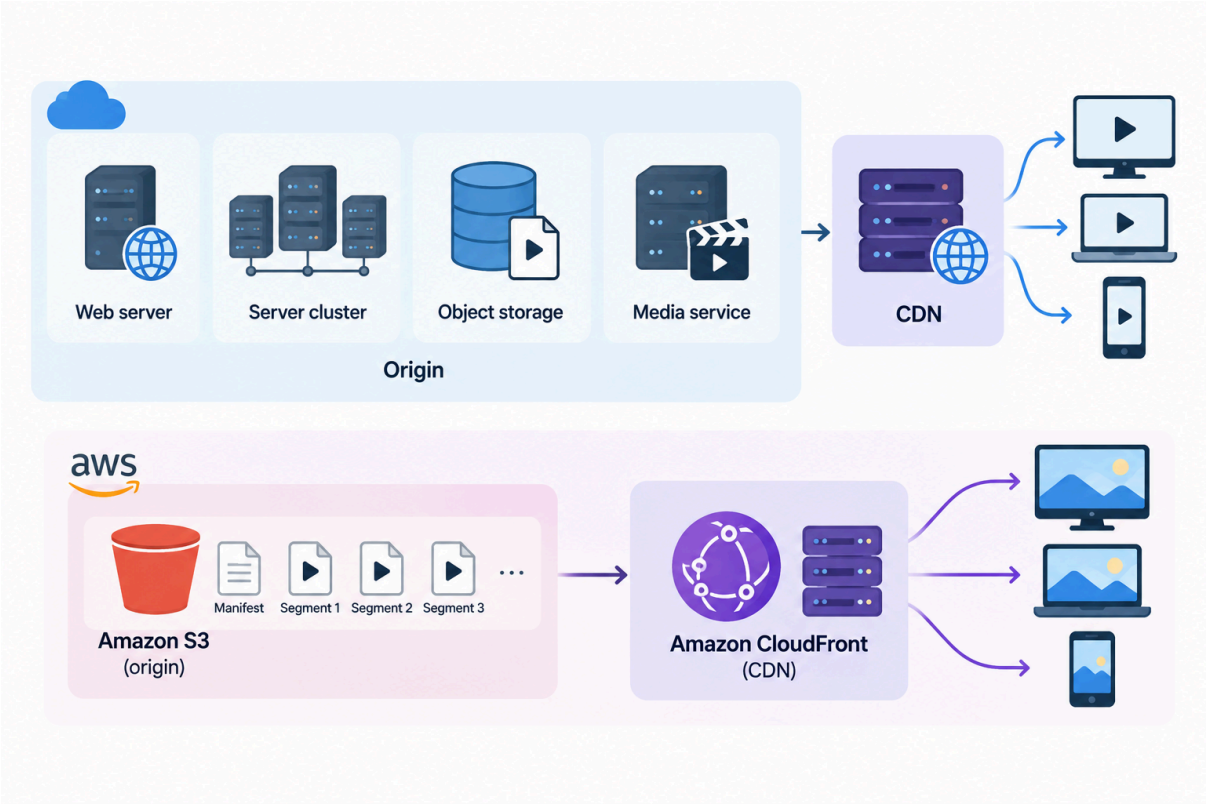
In a modern cloud infrastructure, the two functions can be assigned to different services.

As discussed in the chapter on *CDNs*, the origin is the source from which the *CDN* can retrieve the content it needs to distribute.
The origin can be located within a cloud infrastructure.
However, it does not necessarily have to correspond to a single server.
It may be, for example:

- A web server.

- A server cluster.
- An object storage service.
- A specialised media delivery service.

Amazon, for example, presents Video on Demand architectures in which files are stored in Amazon S3, and the S3 storage containing manifests and segments is used as the origin for Amazon CloudFront (Amazon Web Services, n.d.-b).



One of the most resource-intensive operations in streaming is transcoding.
Different encodings can be produced from an original piece of content, the video master:

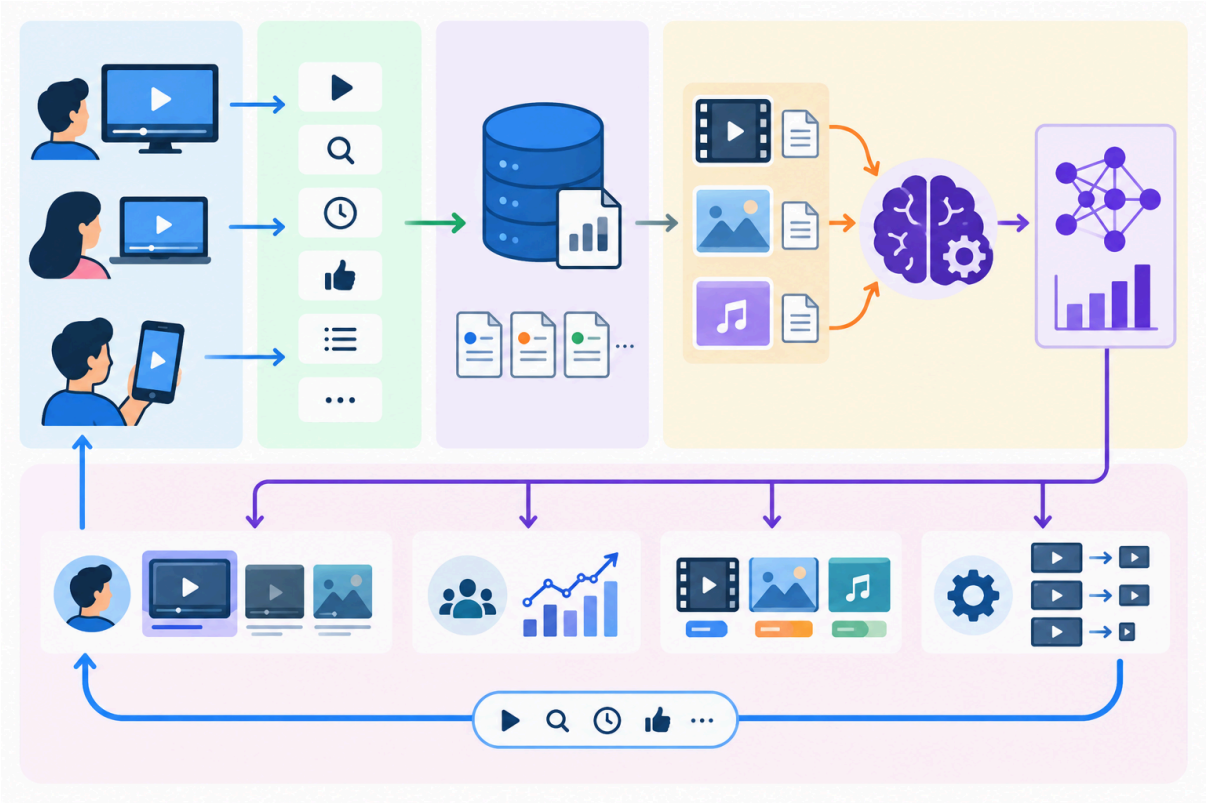
- 1080p → 6 Mbit/s.
- 720p → 3 Mbit/s.
- 480p → 1 Mbit/s.

These operations can be performed using cloud computing resources.
There are also dedicated services, such as *AWS* Elemental MediaConvert, designed to process video content and generate outputs for, among other things, *HLS*, *DASH* and other *ABR* systems (Amazon Web Services, n.d.-g).
The advantage of the cloud is that processing capacity can be increased through elasticity as the workload grows.
For example, a platform that needs to prepare thousands of new videos can temporarily use far more computing resources than during normal periods.

Cloud infrastructures can also be used to run machine learning systems that work on data produced by platforms.
Applications may include:

- Recommendation systems.
- Analysis of user preferences and behaviour.
- Automatic content classification.
- Service quality analysis.
- Optimisation of certain encoding and processing operations.

Platforms can collect user interactions, such as views, searches, viewing time and other actions, as events.
This data is stored and subsequently processed to derive features that machine learning systems can use. Models can then combine user behaviour with content information and other signals to generate personalised recommendations. New interactions in turn produce new data, creating a continuous process of updating and personalisation (Amazon Web Services, n.d.-e).



Netflix, for example, uses AWS for numerous workloads; AWS documents the use of Amazon EMR to process data at scale and personalise recommendations and decisions (Amazon Web Services, n.d.-d).
AWS also documents that Netflix operates across multiple cloud regions and uses autoscaling systems to handle significant load variations (Amazon Web Services, n.d.-d).

Media Player

The media player is the software component that manages the reception and playback of audio or video content on the user's device.

It can be integrated into a browser, a mobile or desktop application, a smart TV, a console or any other device capable of playing multimedia content.

In streaming, the player does more than simply display video.
It is the client-side component that coordinates many of the techniques analysed earlier, deciding which data to request, managing the data received and preparing content for playback.
In *HLS* or *DASH* systems, its operation can be summarised as follows:
Manifest → Segment requests → Buffer → Decoding → Playback.

This process may also include functions such as Adaptive Bitrate Streaming, *DRM*, subtitles, multiple audio tracks and audio/video synchronisation.



When streaming based on technologies such as *HLS* or *DASH* starts, the player initially retrieves the relevant manifest.
In *HLS*, this may be a .m3u8 playlist, while *DASH* uses an .mpd manifest.
Through this information, the player can learn, for example, the available versions of the content, their bitrates and resolutions, the audio tracks, subtitles and media segment addresses (Pantos & May, 2017; DASH Industry Forum, 2018).
The player can then begin requesting the necessary segments:
Manifest → Segment 1 → Segment 2 → Segment 3 → …

When a *CDN* is present, these requests are normally directed to the *CDN* infrastructure and served by an edge server, without the player needing to know whether the content came directly from the cache or was retrieved from the origin.
From the player's perspective, the operation simply involves requesting the resources specified in the manifest.

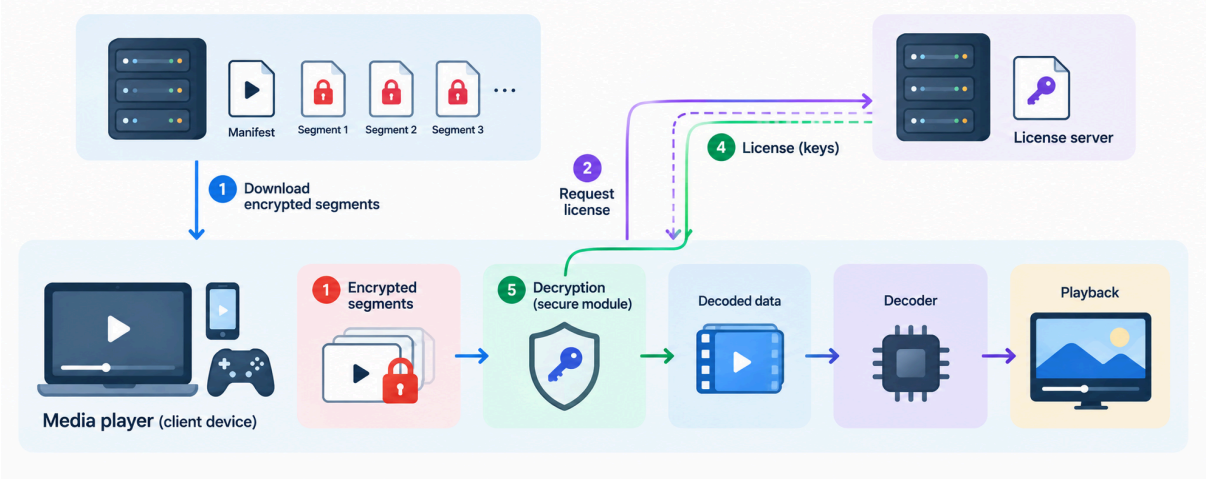
Received segments are not necessarily played immediately.

The player temporarily stores a certain amount of content in the buffer.

The player tries to keep enough content available so that temporary network slowdowns do not immediately interrupt playback.
At the same time, it must avoid unnecessarily accumulating excessive amounts of data, particularly in live systems, where a very large buffer would increase latency.

In systems using *ABR,* the player also performs another fundamental task: it decides which version of the segments to request.
The player therefore continually evaluates several factors, such as observed download speed, the amount of content in the buffer, available bitrates and device characteristics (DASH Industry Forum, n.d.-a).
The aim is to maintain the best possible balance between:
Quality ↔ Continuity ↔ Latency.

If content is protected by a *DRM* (Digital Rights Management) system, media segments may be distributed in encrypted form.
The player can therefore download manifests and segments normally, but cannot directly play the data received.
When it detects that the content is protected, it initiates a request to the service's licence server.
The server checks that the user or device is authorised and, if the check succeeds, returns a licence containing the information needed to access the decryption keys.
Decryption is normally handled by a secure component of the browser, operating system or device, which makes the content available to the decoder without directly exposing the keys to the user (Microsoft, 2026).



Received segments contain audio and video compressed using codecs such as H.264, *HEVC*, VP9, *AV1*, *AAC* or Opus.
Before they can be played, they must be decoded.
The decoder therefore converts compressed data into the information needed to produce images and sounds.

Decoding can be performed through software or specialised hardware components within the device.
Hardware decoding generally reduces CPU load and energy consumption, a particularly important feature on smartphones, tablets and smart TVs (Dutton, 2015).

Audio and video can be managed as separate streams.
The player must therefore ensure that they remain synchronised throughout playback.
To do so, it uses timing information associated with the media data, indicating when a particular video frame or audio sample should be played (Schulzrinne et al., 2003).
The player therefore tries to keep audio ↔ video aligned on the same timeline.
This timing is also particularly important when the user pauses content, seeks to another point in the video or changes playback speed.

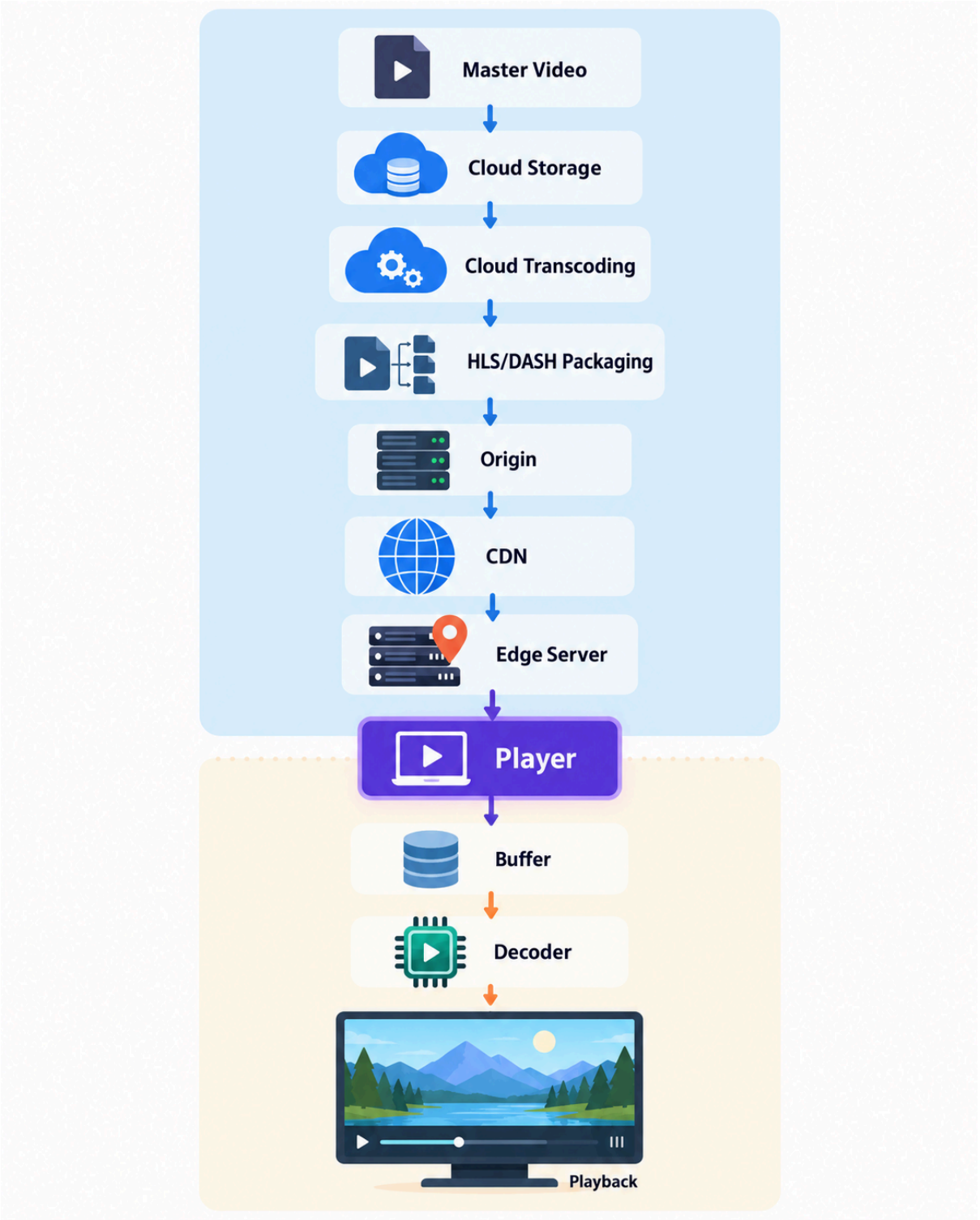
Finally, the media player manages the interface through which the user controls the content.
It can offer functions such as play and pause, fast-forward and rewind, volume adjustment, quality selection, language selection, subtitles, fullscreen mode and playback speed.
When the user moves to another point on the timeline, for example, the player must identify the segments corresponding to the new position and request them from the *CDN*.

The player therefore represents the meeting point between the technical streaming infrastructure and the user's experience.

- Cloud servers process content.
- The origin makes it available.
- The *CDN* distributes it, and edge servers bring it closer to users.
- The player uses all these resources to transform received data into continuous, interactive playback.

We can now bring together the components analysed in the previous chapters.
For on-demand content, we might have:

```
Video master
↓
Cloud Storage
↓
Cloud Transcoding
↓
HLS/DASH Packaging
↓
Origin
↓
CDN
↓
Edge Server
↓
Player
↓
Buffer → Decoder → Playback
```



In this model, each component therefore has a specific function:

- Cloud → stores and processes.
- Origin → makes content available for distribution.
- *CDN* → distributes content geographically.
- Edge → serves requests near users.
- Player → selects, receives, buffers, decodes and plays content.

Supporting material

Exercises

Exercise 1 — Buffering: continuity or latency?
A school is live-streaming a debate between students. During the technical rehearsal, three player configurations are tested:

Configuration	Initial buffer
A	1 second
B	5 seconds
C	15 seconds

The connection is normally sufficient, but occasionally slows down for 3–4 seconds.

- Explain the function of the buffer during playback.
- Which of the three configurations should best withstand a temporary network slowdown? Justify your answer.
- Which should produce the lowest latency?
- Identify the main disadvantage of Configuration A.
- Identify the main disadvantage of Configuration C.
- During transmission, the player receives data faster than it plays it. What happens to the buffer?
- What happens instead if, for an extended period: Data received < data consumed?
- Explain the difference between initial buffering and rebuffering.
- A second problem concerns a connection where all packets arrive, but at very irregular intervals. Is the main problem the amount of data or jitter?
- What function could a jitter buffer perform?
- Why could increasing the jitter buffer too much be problematic during a real-time conversation?
- Choose a configuration for:
    - An on-demand film.
    - A live sports transmission.
    - A videoconference.
- You do not necessarily have to choose the same buffer for all three cases. Justify each decision.
- Conclude by explaining why there is no ideal buffer size valid for every streaming service.

Exercise 2 — Adaptive Bitrate Streaming: which version would you choose?
A platform streams a documentary available through this bitrate ladder:

Version	Resolution	Bitrate

A	2160p	15 Mbit/s
B	1080p	6 Mbit/s
C	720p	3 Mbit/s
D	480p	1 Mbit/s

During viewing, the player detects the following situation:

Segment	Recent network speed	**Available buffer**
1	18 Mbit/s	12 s
2	8 Mbit/s	10 s
3	4 Mbit/s	5 s
4	2 Mbit/s	2 s
5	7 Mbit/s	8 s

- Choose one of the four available versions for each segment.
- Justify each choice, considering both the network and the buffer.
- Why would it be unwise to automatically choose the version whose bitrate is closest to the measured speed?
- What is the main aim of Adaptive Bitrate Streaming?
- Explain why ABR may prefer 720p even when the network temporarily seems sufficient for 1080p.
- What is the relationship between ABR and buffering?
- Why must all renditions be divided into temporally corresponding segments?
- What function does the manifest serve?
- Distinguish between resolution and bitrate. Why might two videos, both at 1080p, not require the same amount of data?
- Now imagine that the viewer is watching on a smartphone with a very small screen. Should the player consider only the available bandwidth? Justify your answer.
- What problem might an algorithm cause if it constantly switched between 480p and 1080p every few seconds?
- Propose a strategy that prioritises continuity first and quality second.
- Conclude by reconstructing the operating cycle: Measurement → quality selection → segment download → playback → new measurement.

Exercise 3 — CDN: distributing a world premiere
An independent platform will release a highly anticipated short film at 21:00.
An audience of 200,000 simultaneous viewers is expected across Europe, Asia and North America.
The initial architecture is: Origin server → users.

- Explain what problem might arise when all users start requesting manifests and segments from the same origin.
- Design a new architecture using a CDN.
- Correctly include:
    - Origin server.
    - PoP.
    - Edge server.
    - User.

- Represent the general path: Origin → … → … → User.
- Explain the difference between a PoP and an edge server.
- Why does the CDN not necessarily have to direct the user to the geographically closest server?
- A user requests segment-24. The edge already has it. Describe the process.
- Another user requests segment-25, which is not present at the edge. Describe the process.
- Then distinguish between a cache hit and a cache miss.
- Why might the second user requesting segment-25 receive it faster?
- Explain why the segmentation used by HLS and DASH is particularly well suited to caching through CDNs.
- Before the premiere, the platform already knows the content and expects very high demand. Compare:
    - Pull caching.
    - Pre-positioning/push.
- Which strategy might be useful in this case? Justify your answer.
- Now imagine a live transmission: why can future segments not be preloaded?
- Does a CDN automatically eliminate every possibility of buffering? Justify your answer.
- Conclude by identifying at least four advantages a CDN offers compared with direct distribution from a single origin.

Exercise 4 — Cloud computing: preparing a platform that needs to scale
A video platform dedicated to short films normally receives 20 new films per day.
During a festival, however, it will receive 2,000 in 48 hours.
For each film, it must:
- Store the original master.
- Generate 1080p, 720p and 480p versions.
- Prepare HLS/DASH manifests and segments.
- Store content ready for distribution.
- Serve users through a CDN.
- Record viewing data.

Design an architecture based on cloud computing.
- Distinguish which operations primarily require storage and which require compute.
- Where would you store the original masters?
- What type of resource would you use to perform transcoding?
- Where would you place manifests and segments after preparation?

- Explain why storage and compute do not necessarily have to be managed by the same service.
- The platform normally needs 5 computing units for transcoding, but temporarily needs 80 during the festival. Which cloud property makes it possible to handle this variation?
- Then define the concept of elasticity.
- Why might permanently purchasing infrastructure sized for peak demand be inefficient?
- Explain why an origin hosted in the cloud does not necessarily have to correspond to a single server.
- Reconstruct a possible workflow: Master → Cloud Storage → … → Packaging → Origin → CDN → Edge → Player.
- Identify which components process content and which primarily handle distribution.
- Explain the difference between:
    - Cloud.
    - CDN.
    - Edge server.
- Why does a CDN remain useful even if the entire platform already uses cloud services?
- Imagine that the number of new pieces of content returns to normal after the festival. What should happen to the computing resources?
- Viewing data is subsequently used to build personalised recommendations. In which part of the architecture might this processing take place?
- Conclude by reconstructing the general model: Cloud / Origin → CDN → Edge Server → User.

Bibliography

Amazon Web Services. (s.d.-a). *Customize at the edge with functions*.

Amazon Web Services. (s.d.-b). *Deliver video on demand with CloudFront*.

Amazon Web Services. (s.d.-c). *How CloudFront delivers content*.

Amazon Web Services. (s.d.-d). *Netflix on AWS*.

Amazon Web Services. (s.d.-e). *What is Amazon Personalize?*.

Amazon Web Services. (s.d.-f). *What is Amazon S3?*.

Amazon Web Services. (s.d.-g). *What is AWS Elemental MediaConvert?*.

Amazon Web Services. (s.d.-h). *What is Amazon IVS Low-Latency Streaming?*.

DASH Industry Forum. (2018). *Guidelines for Implementation: DASH-IF Interoperability Points. Versione 4.3*.

DASH Industry Forum. (s.d.-a). *ABR Settings*.

DASH Industry Forum. (s.d.-b). *Buffer Management*.

Dutton, S. (2015). *High performance video with hardware decoding*.

Fielding, R., Nottingham, M., & Reschke, J. (a cura di). (2022). *HTTP Caching. RFC 9111*.

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing. NIST SP 800-145*.

Microsoft. (2026). *Simple End to End System. Documentazione PlayReady*.

Netflix. (s.d.). *Open Connect Overview*.

Pantos, R., & May, W. (2017). *HTTP Live Streaming. RFC 8216*.

Schulzrinne, H., Casner, S., Frederick, R., & Jacobson, V. (2003). *RTP: A Transport Protocol for Real-Time Applications. RFC 3550*.

W3C. (2025). *Identifiers for WebRTC's Statistics API. Candidate Recommendation Draft, 25 settembre 2025*.

Further reading

- James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach*, 9th Edition, Pearson, 2026. — A comprehensive university textbook examining the principles of computer networking, from Internet applications to transport protocols and network infrastructures. The authors investigate network performance, multimedia applications, content distribution, congestion and the mechanisms supporting modern Internet services. The book is particularly relevant to the handout's discussion of buffering, latency, jitter and Content Delivery Networks because it provides the theoretical foundations needed to understand how bandwidth, throughput, packet loss and network delays influence streaming performance. It also explains why the quality of multimedia delivery depends on the interaction between application-level techniques and the underlying communication infrastructure, offering a broader perspective on the technical trade-offs involved in designing reliable streaming systems.
- DASH Industry Forum, *Buffer Management*, dash.js Documentation. — A technical guide examining how media players manage the temporary storage of audiovisual segments before playback. The documentation describes the configuration of initial buffering, forward buffering and previously downloaded content, explaining how different buffer parameters influence playback behaviour. It is particularly relevant to the handout's distinction between initial buffering and rebuffering, as well as its discussion of the relationship between buffer size, playback continuity and latency. It provides a practical foundation for understanding why increasing the amount of buffered content can improve resistance to temporary network fluctuations while potentially introducing additional delays, especially in live streaming. The guide also demonstrates how buffer management can be configured within a real adaptive streaming player.
- Henning Schulzrinne, Stephen Casner, Ron Frederick and Van Jacobson, *RTP: A Transport Protocol for Real-Time Applications*, RFC 3550, IETF, 2003. — A foundational technical specification defining the Real-time Transport Protocol and its associated control mechanisms for audio and video communication over IP networks.

The document examines packet sequence numbers, timestamps, reception statistics and the calculation of interarrival jitter, providing a rigorous explanation of how timing variations can be measured during multimedia transmission. It is particularly relevant to the handout's distinction between latency and jitter and its discussion of jitter buffers in real-time communication. The specification explains why irregular packet arrival times can affect playback even when packets are eventually delivered, offering the theoretical basis for understanding how receivers estimate network timing variations and why maintaining regular audiovisual playback requires additional buffering and timing mechanisms.

- DASH Industry Forum, *Adaptive Bitrate Streaming*, dash.js Documentation. — A technical investigation of how adaptive streaming players select among different representations of the same audiovisual content. The documentation describes the principles of bitrate adaptation and introduces algorithms that consider estimated network throughput, available buffer levels and device characteristics when selecting the next media segment. It is particularly relevant to the handout's explanation of Adaptive Bitrate Streaming and the relationship between ABR and buffering. The guide provides concrete examples of throughput-based and buffer-based adaptation, including the BOLA algorithm, showing why the player should not simply choose the highest bitrate currently supported by the connection. It also demonstrates how adaptation strategies can be configured to balance image quality, playback continuity and the risk of rebuffering.
- Amazon Web Services, *How CloudFront Delivers Content*, Amazon CloudFront Developer Guide. — A technical explanation of how a Content Delivery Network distributes digital resources through geographically dispersed edge locations and regional caches. Amazon describes how requests are routed to suitable Points of Presence, how cached resources are delivered and how missing content is retrieved from origin servers. The document is particularly relevant to the handout's discussion of CDN architecture, including the distinction between PoPs and edge servers and the mechanisms of cache hits and cache misses. It provides a practical foundation for understanding how geographically distributed caching reduces repeated requests to central infrastructure and improves the scalability of multimedia distribution. It also clarifies why selecting an appropriate CDN node depends on network conditions rather than geographical distance alone.
- Roy T. Fielding, Mark Nottingham and Julian Reschke, *HTTP Caching*, RFC 9111, IETF, 2022. — A technical specification defining how HTTP caches store, validate and reuse responses to requests. The document examines cache freshness, expiration, validation, storage restrictions and the conditions under which previously retrieved resources may be delivered without contacting the origin server again. It is particularly relevant to the handout's explanation of edge caching, cache hits and cache misses in Content Delivery Networks. The specification provides a rigorous foundation for understanding why the presence of an object in a cache does not automatically mean that it can be reused and how HTTP caching rules influence the distribution of manifests, media segments and other streaming resources. It also helps explain the relationship between caching efficiency, network traffic reduction and content consistency.
- Netflix, *Open Connect*, Official Netflix Open Connect Documentation. — An introduction to Netflix's proprietary content delivery infrastructure, developed to distribute audiovisual content efficiently through cooperation with Internet Service

Providers. Netflix explains how Open Connect Appliances can be deployed within ISP networks and how network interconnection arrangements help localise traffic and improve content delivery. The resource is particularly relevant to the handout's comparison between pull caching and content pre-positioning, showing how a streaming provider can use dedicated caching infrastructure to make frequently requested content available closer to viewers. It also offers a concrete example of how Content Delivery Networks can be designed specifically for large-scale video distribution, complementing the more general CDN architecture described in the handout.

- Peter Mell and Timothy Grance, *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, National Institute of Standards and Technology, 2011. — A foundational institutional publication defining the essential characteristics, service models and deployment models of cloud computing. The authors identify properties such as on-demand self-service, resource pooling, rapid elasticity, broad network access and measured service, explaining how cloud infrastructures differ from traditional computing environments. The document is particularly relevant to the handout's discussion of cloud computing, especially its explanation of elasticity and the distinction between storage and processing resources. It provides the theoretical framework for understanding why a streaming platform can dynamically allocate additional computing capacity to transcoding workloads, manage large multimedia libraries through cloud storage and reduce resource consumption when demand decreases.
- Amazon Web Services, *Video on Demand and Live Streaming Video with CloudFront*, Amazon CloudFront Developer Guide. — A technical overview of cloud-based architectures used to prepare and distribute prerecorded and live audiovisual content. The documentation examines how encoding, packaging, origin infrastructure and CDN services collaborate to deliver media segments and manifests to viewers. It is particularly relevant to the handout's description of the relationship between cloud computing and Content Delivery Networks because it demonstrates how cloud storage and processing services can be integrated with geographically distributed content delivery. The resource also provides concrete examples of different workflows for on-demand and live streaming, helping explain why cloud computing and CDN distribution perform complementary functions rather than representing interchangeable technologies.
- MDN Web Docs, *Media Source API*, Mozilla Developer Network. — A technical introduction to Media Source Extensions, a browser technology that allows web applications to manage multimedia data dynamically during playback. The documentation describes how JavaScript applications can supply media segments to audio and video elements through MediaSource and SourceBuffer objects, providing the foundation for adaptive streaming players. It is particularly relevant to the handout's section on Media Players because it explains how a player can receive, buffer and organise media data without relying on the progressive download of a single complete file. The guide also clarifies the relationship between browser playback capabilities and adaptive streaming technologies such as MPEG-DASH, showing how client-side software coordinates segment delivery, buffer management and continuous audiovisual presentation.

Editorial notes

In producing these materials, I have used generative artificial intelligence tools to support the writing process, particularly to improve the form of the text, reorganize content, refine wording, and speed up certain editorial tasks.

As I work independently on the production of these materials, I try to automate any activities that can reasonably be automated, so that I can devote more time to research, design, and content development.

However, artificial intelligence does not determine the content of this work: the choice of topics, structure, ideas, interpretations, examples, and teaching approach are developed by me. AI is therefore used primarily as a tool to support production and formal revision, while authorship and responsibility for the design and content remain mine.

Unless otherwise indicated, this material is distributed under the Creative Commons Attribution–NonCommercial–ShareAlike 4.0 International license (CC BY-NC-SA 4.0).

You are therefore free to share, redistribute, adapt, and create derivative works based on this material, provided that appropriate credit is given, the material is not used for commercial purposes, and any modified or derivative versions are distributed under the same license.

