Welcome to Module 11: Caching Content.

Module overview

aws academy

Sections

1. Architectural need
2. Overview of caching
3. Edge caching
4. Caching web sessions
5. Caching databases

Lab

- Guided Lab: Streaming Dynamic Content Using Amazon CloudFront

Knowledge check

This module includes the following sections:

1. Architectural need
2. Overview of caching
3. Edge caching
4. Caching web sessions
5. Caching databases

The module also includes a guided lab in which you will learn how to stream dynamic content by using Amazon CloudFront.

Finally, you will be asked to complete a knowledge check that will test your understanding of key concepts covered in this module.

Module objectives

aws academy

At the end of this module, you should be able to:

- Identify how caching content can improve application performance and reduce latency
- Identify how to design architectures that use edge locations for distribution and distributed denial of service (DDoS) protection
- Create architectures that use Amazon CloudFront to cache content
- Recognize how session management relates to caching
- Describe how to design architectures that use Amazon ElastiCache

At the end of this module, you should be able to:

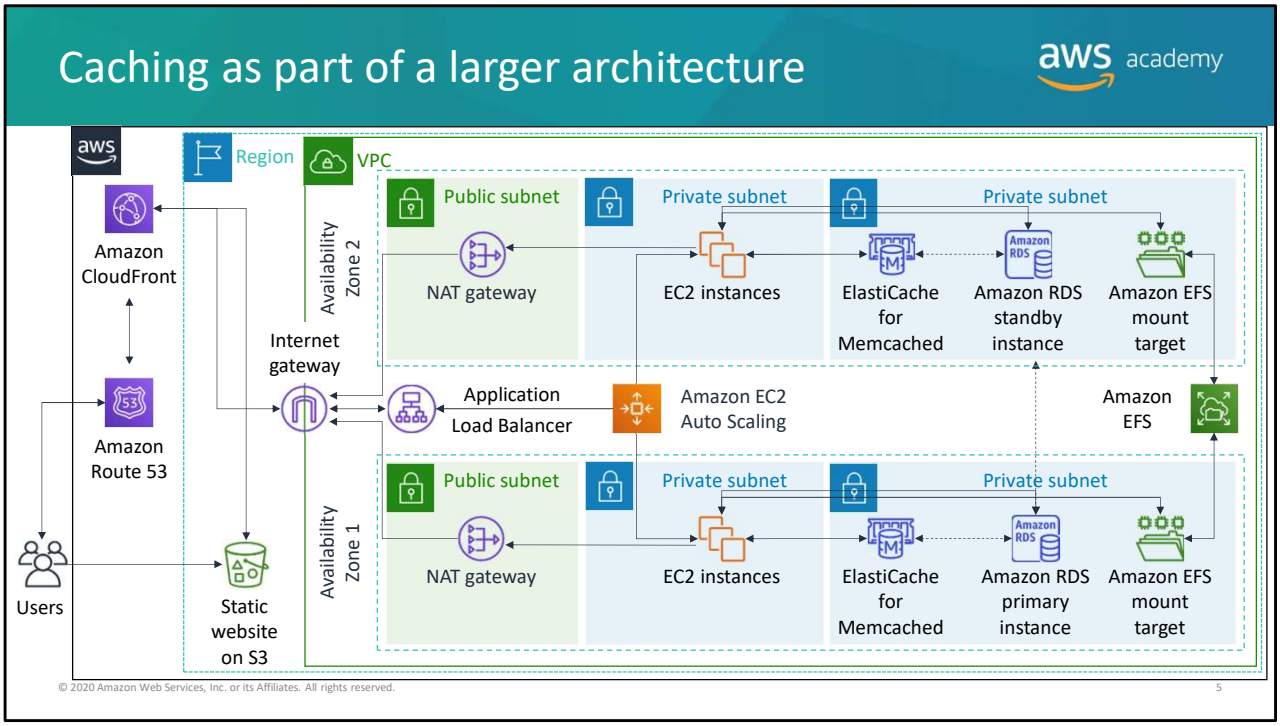
- Identify how caching content can improve application performance and reduce latency
- Identify how to design architectures that use edge locations for distribution and distributed denial of service (DDoS) protection
- Create architectures that use Amazon CloudFront to cache content
- Recognize how session management relates to caching
- Describe how to design architectures that use Amazon ElastiCache

Module 11: Caching Content

Section 1: Architectural need

aws academy

Introducing Section 1: Architectural need.



In this module, you will learn how to implement caching in your networking environment. The final components of the architecture diagram (that is, Amazon ElastiCache and Amazon CloudFront) are introduced in this module, and have been revealed. This module will also cover database caching with Amazon DynamoDB.

Café business requirement

The capacity of the café's infrastructure is constantly being overloaded with the same requests. This inefficiency is increasing cost and latency.

The capacity of the café's infrastructure is constantly being overloaded with the same requests for static content, such as images of menu items. This situation is increasing both cost and latency. Sofía and Nikhil want to identify and cache frequently accessed static content to reduce latency and improve the customer experience. Every time a customer loads the menu, it is served from the cache. However, when menu items change, the request is routed to the database, and the cache is updated.

In addition, local celebrities are starting to endorse the café through video testimonials. Sofía and Nikhil must use edge caching for streaming data so they can serve appropriate, regionally-appealing content based on the geolocation of the user who is loading the site.

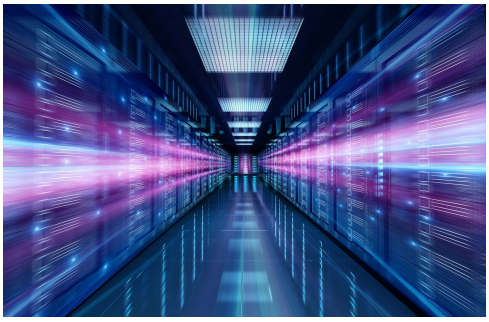
Module 11: Caching Content

Section 2: Overview of caching

aws academy

Introducing Section 2: Overview of caching.

Caching: Trading capacity for speed



- Is a high-speed data storage layer
- Stores a subset of data
- Increases data retrieval performance
- Reduces the need to access the underlying slower storage layer

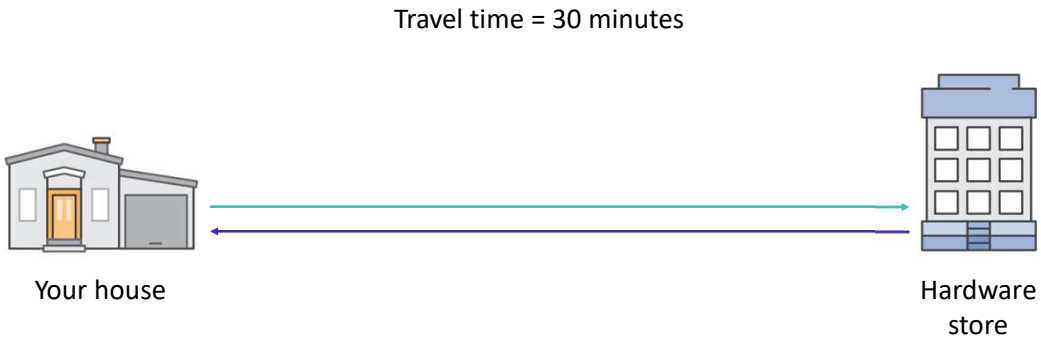
Speed is important, whether your application serves the latest news, a top-10 leaderboard, a product catalog, or sells tickets to an event. The speed at which you deliver content affects the success of your application. If someone wants data, whether for a webpage or a report that drives business decisions, you can deliver that data much faster if it's cached.

In computing, a *cache* is a high-speed data storage layer. Unlike a database, which usually stores data in a complete and durable form, a cache transiently stores a subset of data. The primary purpose of a cache is to increase the performance of data retrieval by reducing the need to access the underlying, slower storage layer. Future requests for cached data are served faster than requests that access the data's primary storage location.

Caching trades capacity for speed, and it enables you to efficiently reuse previously retrieved or computed data.

The data in a cache is generally stored in fast-access hardware, such as random access memory (RAM).

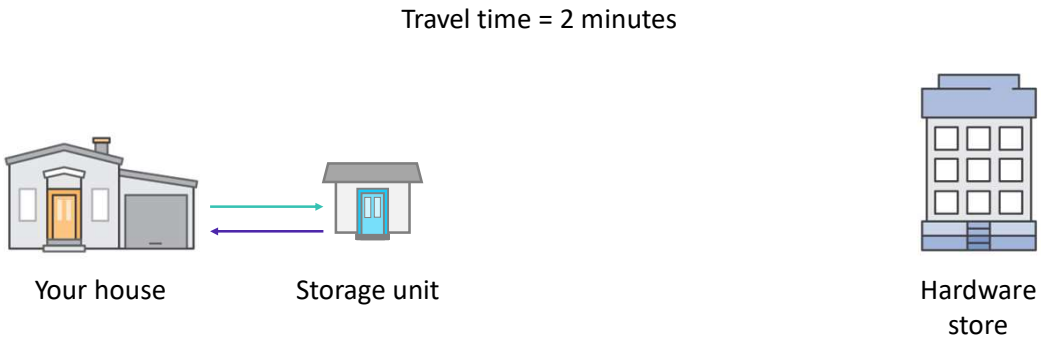
Cache example (1 of 2)



To illustrate how a cache improves performance, consider the example of making a trip to a hardware store.

If the store is miles away, it takes you considerable effort to go there every time you need something.

Cache example (2 of 2)



Instead, you can store the supplies that you use regularly in a storage unit close to your house. Thus, it takes you less time to access these supplies than it would to go all the way to the hardware store.

However, you could still go to the store to refresh your supplies.

In this example, the storage unit is analogous to a cache.

What should you cache?

Data that requires a slow and expensive query to acquire

Relatively static and frequently accessed data—for example, a profile for your social media website

Information that can be stale for some time, such as a publicly traded stock price

When you decide what data to cache, consider these factors:

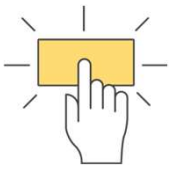
Speed and expense – Time-consuming database queries and frequently used, complex queries often create bottlenecks in applications. Generally, if the data requires a slow and expensive query to acquire, it's a candidate for caching. For example, queries that perform joins on multiple tables are slower and more expensive than simple queries on single tables. However, even data that requires a relatively quick and simple query might still be a candidate for caching, depending on other factors.

Data and access patterns – Determining what to cache also involves understanding the data itself and its access patterns. For example, it doesn't make sense to cache webpages that return search results that are unusually dynamic in nature. For caching to provide meaningful benefits, the data should be relatively static and frequently accessed, such as a personal profile on a social media site. Conversely, you don't want to cache data if caching it provides no speed or cost advantage. For example, it doesn't make sense to cache webpages that return the results of a search because these queries and results are almost always unique.

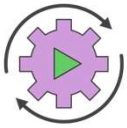
Staleness – By definition, cached data is stale data. Even if it’s not stale in certain circumstances, cached data should be considered and treated as stale. When you determine whether your data is a candidate for caching, you must also determine your application's tolerance for stale data. Your application might be able to tolerate stale data in one context, but not another. For example, consider an application that serves a publicly traded stock price on a website. With this application, a short delay might be acceptable if it has a disclaimer that prices might be delayed up to *n* minutes. But when an application must serve the stock price to a broker who is making a sale or purchase, you want real-time data.

Benefits of caching

Improves application speed



Reduces response latency



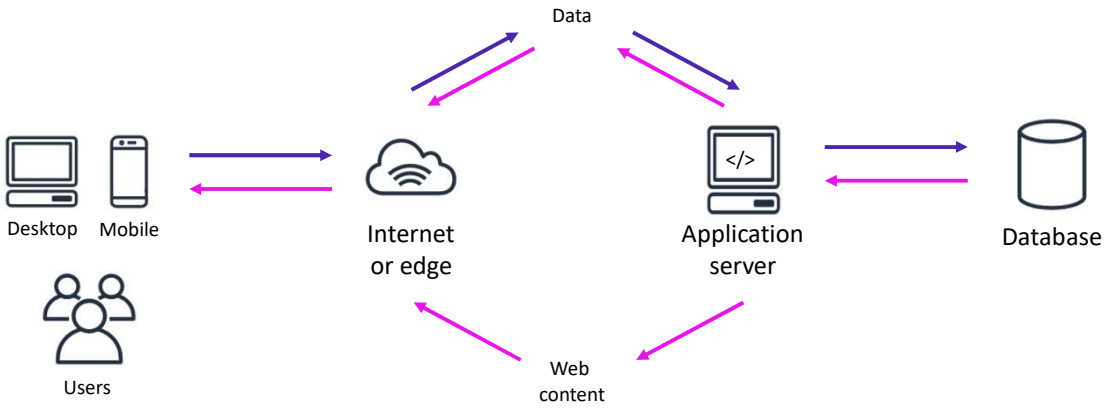
Reduces database access time

A cache provides high throughput, low-latency access to commonly accessed application data by storing the data in memory.

Caching can:

- Improve the speed of your application.
- Reduce the response latency that users experience with your application.
- Reduce application processing time and database access time for read-heavy workloads, such as social networking, gaming, media sharing, and Q&A portals. Write-heavy applications typically do not see as great a benefit from caching. However, even write-heavy applications normally have a read/write ratio greater than 1, which implies that read caching can still be beneficial.

Caching throughout the data journey



This simple web application architecture shows how data flows from and to the user. You can use caching at each layer to improve the overall performance and usability of your application. These layers include operating systems, networking layers like content delivery networks (CDNs) and Domain Name Systems (DNS), web applications, and databases.

In this architecture, you can use caching to:

- Accelerate retrieval of information from websites
- Store a mapping of domain names to IP addresses
- Accelerate retrieval of web content from web servers or application servers
- Accelerate application performance and data access
- Reduce latency that is associated with database query requests

In this module, you will learn how different AWS services support caching at these various layers.

Section 2 key takeaways

- A cache provides high throughput, low-latency access to commonly accessed application data by storing the data in memory
- When you decide what data to cache, consider speed and expense, data and access patterns, and your application's tolerance for stale data
- Caches can be applied and used throughout various layers of technology, including operating systems, networking layers, web applications, and databases

Some key takeaways from this section of the module include:

- A cache provides high throughput, low-latency access to commonly accessed application data by storing the data in memory
- When you decide what data to cache, consider speed and expense, data and access patterns, and your application's tolerance for stale data
- Caches can be applied and used throughout various layers of technology, including operating systems, networking layers, web applications, and databases

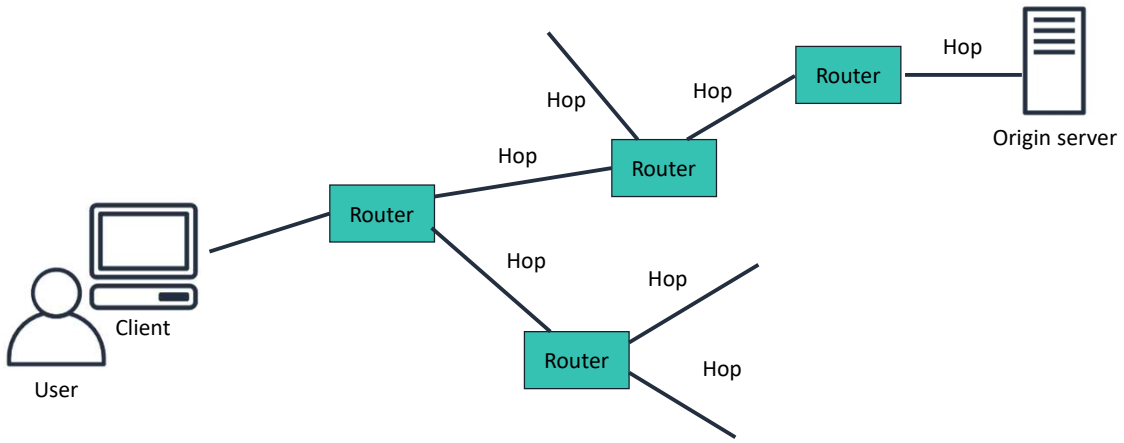
Module 11: Caching Content

Section 3: Edge caching

aws academy

Introducing Section 3: Edge caching.

Network latency



When someone browses to your website or uses your application, their request is routed through many different networks to reach your origin server. The origin server—which is also referred to as *origin*—stores the original, definitive versions of your objects (for example, web objects, images, and media files). The number of network hops and the distance that the request travels can significantly affect the performance and responsiveness of your website.

Further, network latency can depend on the geographic location of the origin server. When your web traffic is geographically dispersed, it's not always feasible (or cost-effective) to replicate your entire infrastructure across the globe. In this case, a content delivery network (CDN) can be useful.

Content delivery network (CDN)

- Is a globally distributed system of caching servers
- Caches copies of commonly requested files (static content)
- Delivers a local copy of the requested content from a nearby cache edge or Point of Presence
- Improves application performance and scaling

A content delivery network (CDN) is a globally distributed system of caching servers. A CDN caches copies of commonly requested files that are hosted on the application origin server. These files can include static content, such as HTML, CSS, JavaScript, image, and video files. The CDN delivers a local copy of the requested content from a cache edge or Point of Presence (PoP) that provides the fastest delivery to the requester.

For more information about caching with CDNs, see Content Delivery Network (CDN) Caching.

Amazon CloudFront

- Is the Amazon global CDN
- Is optimized for all delivery use cases, with a multi-tier cache by default and extensive flexibility
- Provides an extra layer of security for your architectures
- Supports WebSockets and HTTP or HTTPS methods

Amazon CloudFront is a global CDN service that accelerates the delivery of content to users. Such content might be static and dynamic content, media files that use HTTP or HTTPS, and streaming video (both video on demand and live streaming). Like other AWS services, CloudFront is a self-service, pay-per-use offering, which requires no long-term commitments or minimum fees.

The CDN offers a multi-tier cache by default. Regional edge caches improve latency and lower the load on your origin servers when the object is not already cached at the edge. In addition, the CDN offers multiple options for streaming your media, both pre-recorded files and live events. It offers them at the sustained, high throughput that is required for 4K delivery to global viewers.

CloudFront provides both network-level and application-level protection. Your traffic and applications benefit through various built-in protections, such as AWS Shield Standard, at no extra cost. You can also use configurable features such as AWS Certificate Manager (ACM) to create and manage custom SSL certificates at no extra cost. CloudFront supports Secure Sockets Layer/Transport Layer Security (SSL/TLS) protocols.

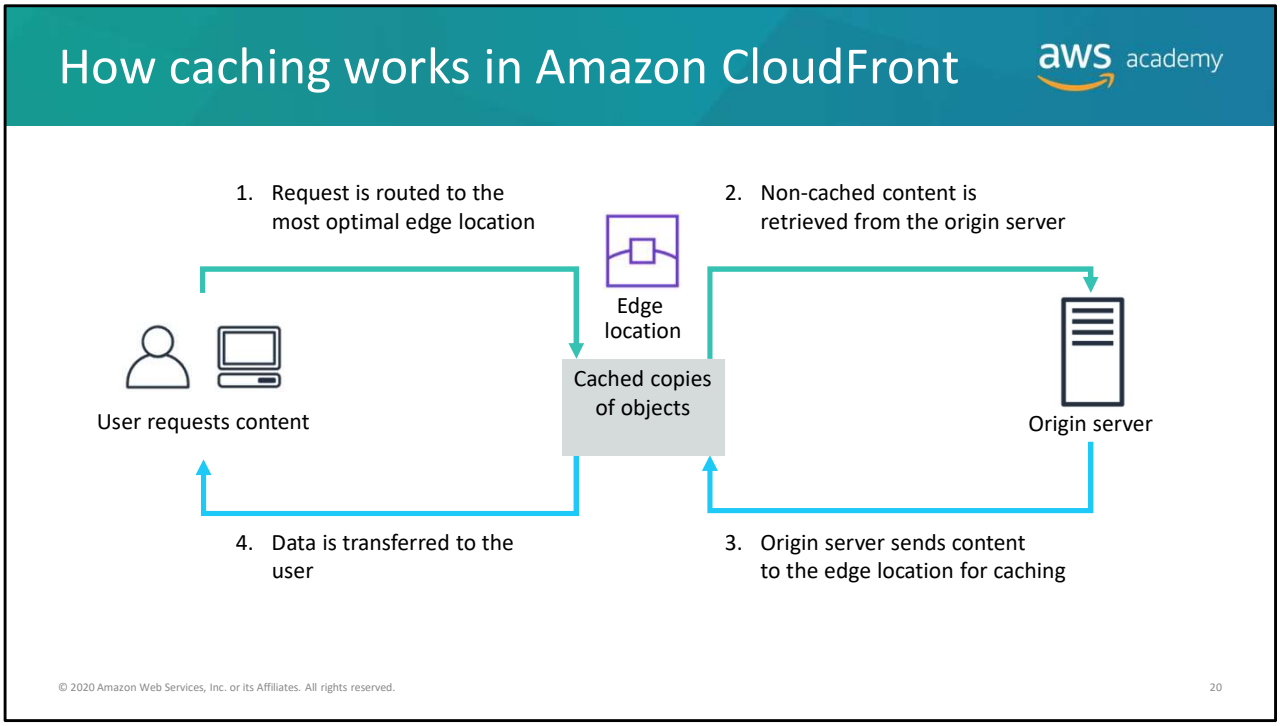
CloudFront supports real-time, bidirectional communication over the WebSocket protocol. This persistent connection enables clients and servers to send real-time data to one another without the cost of repeatedly opening connections. It is especially useful for communications applications, such as chat, collaboration, gaming, and financial trading. CloudFront also supports HTTP methods (DELETE, GET, HEAD, OPTIONS, PATCH POST, PUT), which improve the performance of dynamic websites. These websites have web forms, comment and login boxes, add-to-cart buttons, and other features that upload data from users. Thus, you can use a single domain name to deliver your entire website through CloudFront, which accelerates both the download and upload parts of your website.

What type of content can you cache in an edge cache?

This example of an Amazon.com webpage shows how static and dynamic content can compose a dynamic web application. This web application is delivered through the HTTPS protocol for the encryption of user page requests and the pages that are returned from a web server. You can use a CDN or edge cache to cache static content. Such content might include web objects (for example, HTML documents, CSS style sheets, or JavaScript files), image files, and video files.

You cannot cache dynamically generated content or user-generated data. However, you can configure CloudFront to deliver this information from an application that runs on a custom origin. For example, it might be an EC2 instance or a web server.

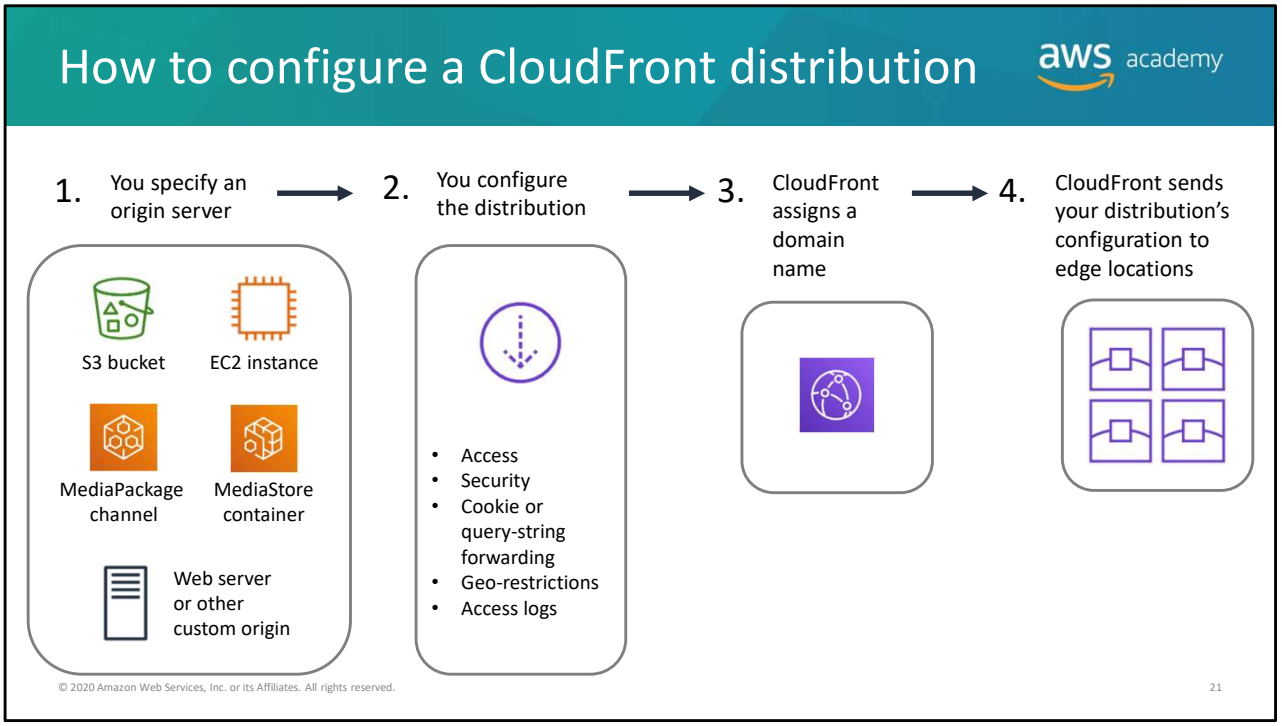
Additionally, you can configure CloudFront to require that viewers use HTTPS to request your objects, so that connections are encrypted when CloudFront communicates with viewers. You also can configure CloudFront to use HTTPS to get objects from your origin, so that connections are encrypted when CloudFront communicates with your origin.



Amazon CloudFront delivers your content to users through a worldwide network of data centers that are called *edge locations*.

When a user requests content that you are serving with CloudFront, DNS routes the request to the edge location that best serves the request. Typically, it is the nearest edge location that has the shortest latency. CloudFront checks the cache for the requested content (step 1). If the content is in the cache, CloudFront delivers it immediately to the user (step 4). The content might not be currently in the cache. If not, CloudFront forwards the request to the origin server that you identified as the source for the definitive version of your content (step 2). The origin server sends the content back to the edge location (step 3), and CloudFront then forwards the content to the user (step 4). It also adds the content to the cache in the edge location for the next time someone requests it.

As objects become less popular, individual edge locations might remove those objects to make room for more popular content. For the less popular content, CloudFront has *regional edge caches.* Regional edge caches are CloudFront locations that are deployed globally and close to your viewers. They are located between your origin server and the global edge locations that serve content directly to viewers. A regional edge cache has a larger cache than an individual edge location, so objects remain in the regional edge cache longer. This arrangement helps to keep more of your content closer to your viewers. It reduces the need for CloudFront to go back to your origin server, and it improves overall performance for viewers.



When you want to use CloudFront to distribute your content, you create a *distribution*.

1. You specify the origin server that hosts your files. Your origin server can be an S3 bucket, an AWS Elemental MediaPackage channel, an AWS Elemental MediaStore container, or a custom origin. For example, a custom origin might be an EC2 instance or your own web server.

2. You then specify details about how to track and manage content delivery. For example, you can specify whether you want your files to be available to everyone or only to certain users. You can also specify whether you want CloudFront to perform the following functions: create access logs that show user activity, forward cookies or query strings to your origin, or require users to use HTTPS to access your content.

3. CloudFront assigns a domain name to your new distribution.

4. CloudFront sends your distribution's configuration—but not the content—to all edge locations.

How to expire content

- Time to Live (TTL) –
  - Fixed period of time (expiration period)
  - Set by you
  - GET request to origin from CloudFront uses If-Modified-Since header
- Change object name –
  - Header-v1.jpg becomes Header-v2.jpg
  - New name forces immediate refresh
- Invalidate object –
  - Last resort: inefficient and expensive

You can expire cached content in three ways:

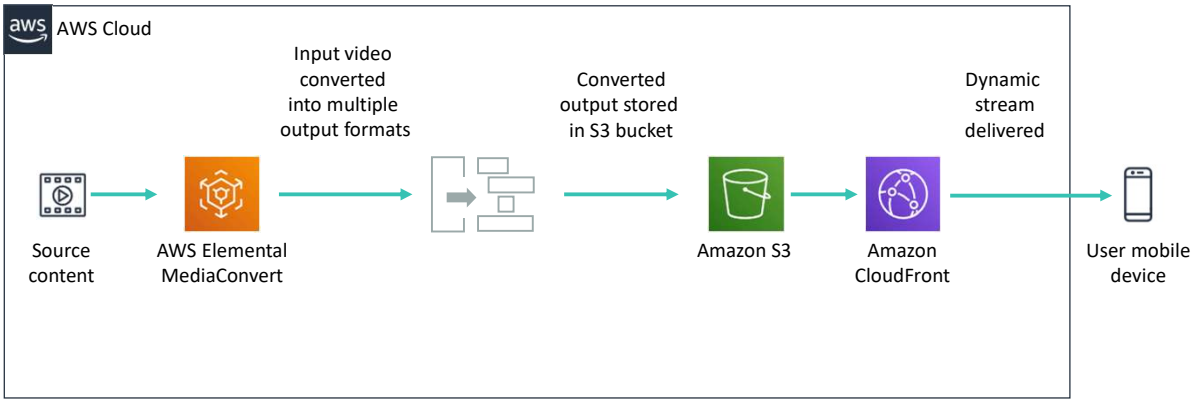
- Time to Live (TTL) – With this method, you can control how long your files stay in a CloudFront cache before CloudFront forwards another request to your origin. Reducing the duration enables you to serve dynamic content. Increasing the duration means that your users get better performance because your files are more likely to be served directly from the edge cache. A longer duration also reduces the load on your origin. If you set the TTL for a particular origin to *0*, CloudFront still caches the content from that origin. It then makes a GET request with an If-Modified-Since header. Thus, the origin has a chance to signal that CloudFront can continue to use the cached content if it hasn't changed at the origin. TTL is a good method to use if the replacement doesn’t need to be immediate.

- Change object name – This method requires more effort, but replacement is immediate. Although you *can* update existing objects in a CloudFront distribution and use the same object names, it is not recommended. CloudFront distributes objects to edge locations only when the objects are requested—not when you put new or updated objects in your origin. For example, you might update an existing object in your origin with a newer version that has the same name. In that case, an edge location won’t get that new version from your origin until both of the listed events occur.

- Invalidate object – This method is a bad solution because the system must forcibly interact with all edge locations. You should use this method sparingly and only for individual objects.

For more information about how to expire cache content, see Managing How Long Content Stays in an Edge Cache (Expiration).

Example: Video on demand streaming



As you have learned, you can use CloudFront to deliver streaming video—both video on demand and live streaming.

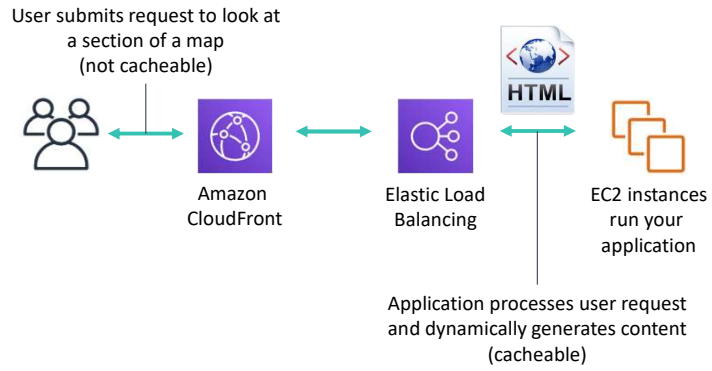
For on-demand video streaming, you must use an encoder to format and package video content before CloudFront can distribute it. Examples of encoders include AWS Elemental MediaConvert and Amazon Elastic Transcoder. The packaging process creates *segments*, which are static files that contain your audio, video, and captions content. It also generates manifest files, which describe what segments to play and the specific order to play them in. Package formats include Dynamic Adaptive Streaming over HTTP (DASH, or MPEG-DASH), Apple HTTP Live Streaming (HLS), Microsoft Smooth Streaming, and Common Media Application Format (CMAF).

After converting the video into the output formats, you host the converted content in an S3 bucket, which is your origin server. You then use CloudFront to deliver the segment files to users around the world.

For more information on video streaming with CloudFront, see Video on Demand and Live Streaming Video with CloudFront.

Example: Dynamically generated content

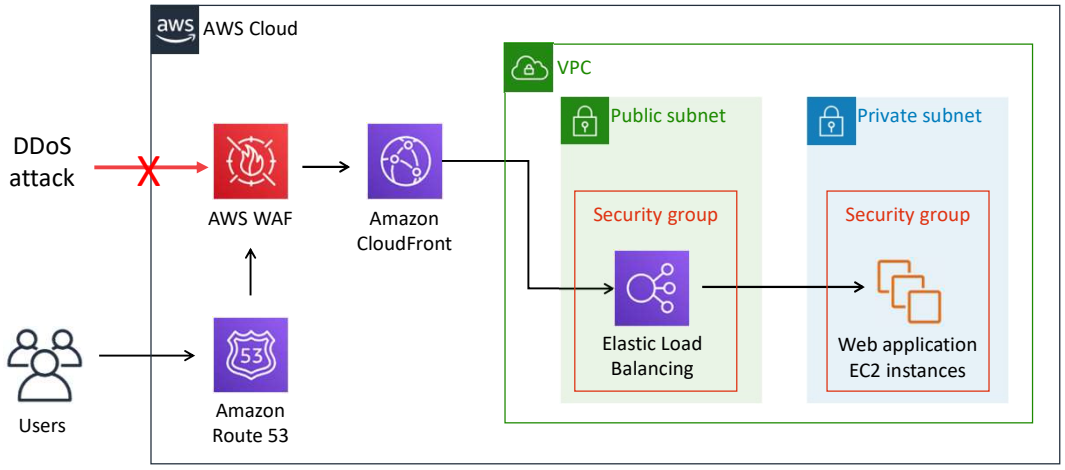
Use case: Map tiles
Problem: Need faster DB response time



In general, you cache only static content. However, you can have content that *appears* static (because of its URL), but which is built dynamically the first time that it's needed. This dynamic build can be useful when the content is reusable, but it can be expensive to create, and it can change infrequently.

Map tiles are the classic example. In this case, you cache places that people frequently view (such as major cities), but not places like remote areas. Generating all possible combinations of tiles would be prohibitively expensive and wasteful because most tiles would almost never be requested. Instead, the path component of each tile's URL can include the parameters needed to generate the tile. If the tile is already present in a particular CloudFront edge location, then it is served up directly. Otherwise, it is generated, returned to the edge location, and then used to satisfy future requests.

Example: DDoS mitigation



You can use CloudFront to improve the resiliency of your applications that run on AWS from distributed denial of service (DDoS) attacks. A DDoS attack is a deliberate attempt to make your website or application unavailable to users, for example, by flooding it with network traffic. To achieve this end, attackers use multiple sources to orchestrate an attack against a target. These sources might include distributed groups of malware-infected computers, routers, Internet of Things (IoT) devices, and other endpoints.

The following example shows a resilient architecture that can help prevent or mitigate DDoS attacks.

A DNS service, such as Amazon Route 53, can effectively connect users' requests to a CloudFront distribution. The CloudFront distribution then proxies requests for dynamic content to the infrastructure that hosts your application's endpoints. Both Route 53 DNS requests and subsequent application traffic that are routed through CloudFront are inspected inline. Always-on monitoring, anomaly detection, and mitigation against common infrastructure DDoS attacks are built into both Route 53 and CloudFront.

Common infrastructure attacks include synchronize/acknowledge (SYN/ACK) floods, User Datagram Protocol (UDP) floods, and reflection attacks. When the SYN flood attack threshold is exceeded, SYN cookies are activated to avoid dropping connections from legitimate clients. Deterministic packet filtering drops malformed TCP packets and invalid DNS requests, and permits traffic to pass only if the traffic is valid for the service. Heuristics-based anomaly detection evaluates attributes such as type, source, and composition of traffic. Traffic is scored across many dimensions, and only the most suspicious traffic is dropped.

This method enables you to avoid false positives while you protect application availability. Route 53 is also designed to withstand DNS query floods. DNS query floods are real DNS requests that can continue for hours and attempt to exhaust DNS server resources. Route 53 uses shuffle sharding and anycast striping to spread DNS traffic across edge locations and help protect the availability of the service.

AWS WAF is a web application firewall. It enables you to monitor the HTTP and HTTPS requests that are forwarded to an Amazon API Gateway API, CloudFront, or Application Load Balancer. AWS WAF also enables you to control access to your content. For example, you can specify conditions such as the IP addresses that requests originate from or the values of query strings. Based on these conditions, API Gateway, CloudFront, or an Application Load Balancer responds with either the requested content or an HTTP 403 status code (Forbidden). You also can configure CloudFront to return a custom error page when a request is blocked.

For more information about how to improve the resiliency of your applications that run on AWS against DDoS attacks, see the following resources:

- AWS Best Practices for DDoS Resiliency AWS whitepaper
- How to Help Protect Dynamic Web Applications Against DDoS Attacks by Using Amazon CloudFront and Amazon Route 53 AWS Security Blog post

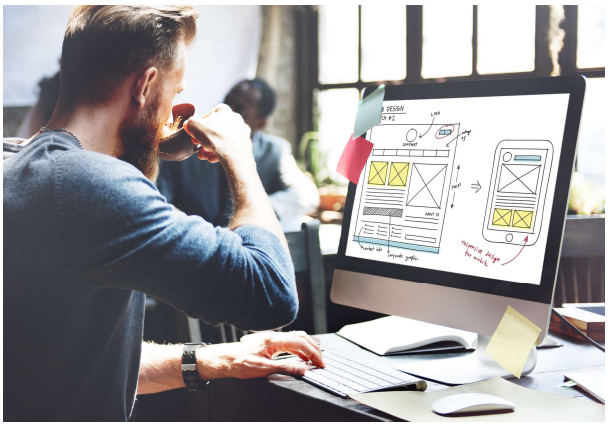
Section 3 key takeaways

- Amazon CloudFront is a global CDN service that accelerates the delivery of content, including static and video, to users with no minimum usage commitments.
- CloudFront uses a global network that comprises edge locations and regional edge caches to deliver content to your users.
- To use CloudFront to deliver your content, you specify an origin server and configure a CloudFront distribution. CloudFront assigns a domain name and sends your distribution's configuration to all of its edge locations.
- You can use Amazon CloudFront to improve the resilience of your applications that run on AWS from DDoS attacks.

Some key takeaways from this section of the module include:

- Amazon CloudFront is a global CDN service that accelerates the delivery of content, including static and video, to users with no minimum usage commitments.
- CloudFront uses a global network that comprises edge locations and regional edge caches to deliver content to your users.
- To use CloudFront to deliver your content, you specify an origin server and configure a CloudFront distribution. CloudFront assigns a domain name and sends your distribution's configuration to all of its edge locations.
- You can use Amazon CloudFront to improve the resilience of your applications that run on AWS from DDoS attacks.

Module 11 – Guided Lab: Streaming Dynamic Content Using Amazon CloudFront



You will now complete Module 11 – Guided Lab: Streaming Dynamic Content Using Amazon CloudFront.

Guided lab: Scenario

In this lab, you use Amazon Elastic Transcoder to convert a source video into multiple bitrates. You use Amazon CloudFront to deliver the dynamic, multiple bitrate stream to a connected device by using Apple HTTP Live Streaming (HLS) protocol.

Amazon Elastic Transcoder

Amazon CloudFront

In this lab, you use Amazon Elastic Transcoder to convert a source video into multiple bitrates. You use Amazon CloudFront to deliver the dynamic, multiple-bitrate stream to a connected device by using Apple HTTP Live Streaming (HLS) protocol. The stream can be played on any browser that supports the HLS protocol.

Apple HLS can dynamically adjust movie playback quality to match the available speed of wired or wireless networks by using an ordinary web server. It works by creating different quality streams. Each stream is then broken into chunks that are streamed sequentially to a client device. On the client's end, you can select streams of varying bitrates, which enable streaming sessions to adapt to different network speeds.

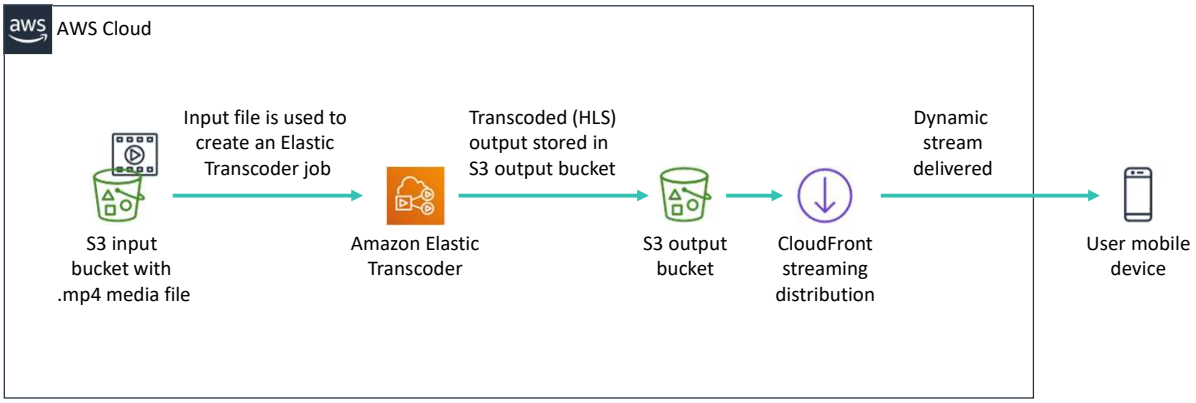
Guided lab: Tasks

1. Create an Amazon CloudFront distribution
2. Create an Amazon Elastic Transcoder pipeline
3. Test playback of the dynamic (multiple bitrate) stream

In this guided lab, you will complete the following tasks:

1. Create an Amazon CloudFront distribution
2. Create an Amazon Elastic Transcoder pipeline
3. Test playback of the dynamic (multiple bitrate) stream

Guided lab: Final product



The diagram summarizes what you will have built after you complete the lab.

~ 30 minutes

It is now time to start the guided lab.

Guided lab debrief: Key takeaways

Your educator might choose to lead a conversation about the key takeaways from this guided lab after you have completed it.

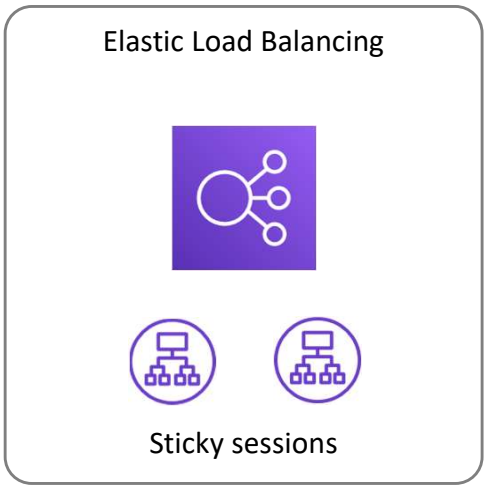
Module 11: Caching Content

Section 4: Caching web sessions

aws academy

Introducing Section 4: Caching web sessions.

Session management: Sticky sessions



Feature that enables a load balancer to route a request to the specific server that manages the user's session.

- Use client-side cookies
- Are cost-effective
- Speed up retrieval of sessions
- Have disadvantages –
  - Loss of sessions when you have an instance failure
  - Limit scalability: Uneven load distribution and increased latency

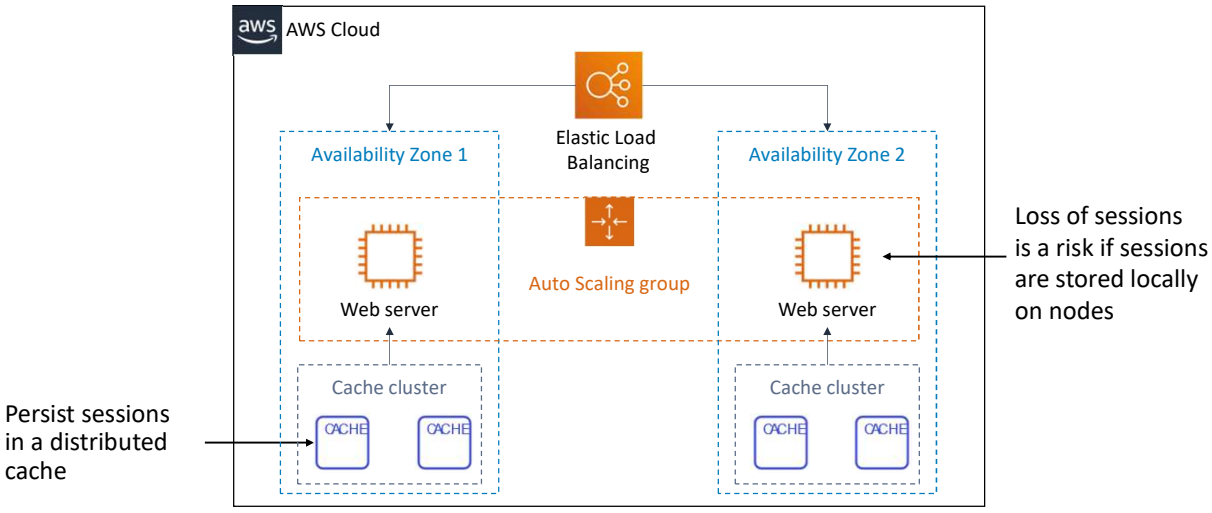
When a user or service interacts with a web application, it sends an HTTP request, and the application returns a response. A sequence of such transactions is called a *session*. Every request is independent of previous transactions. Therefore, sessions are used to manage user authentication and store user data while the user interacts with the application. For example, by using sessions, your users are not required to send their credentials for every request that they make to your server.

You can manage user sessions in various ways. (By default, a load balancer routes each request independently to the registered instance with the smallest load.) To use sticky sessions, the client must support cookies.

Sticky sessions are cost-effective because the sessions are stored on the web servers that are running your applications. Therefore, sticky sessions eliminate network latency and speed up retrieval of those sessions. However, in the event of instance failure, you are likely to lose the sessions that are stored on that instance.

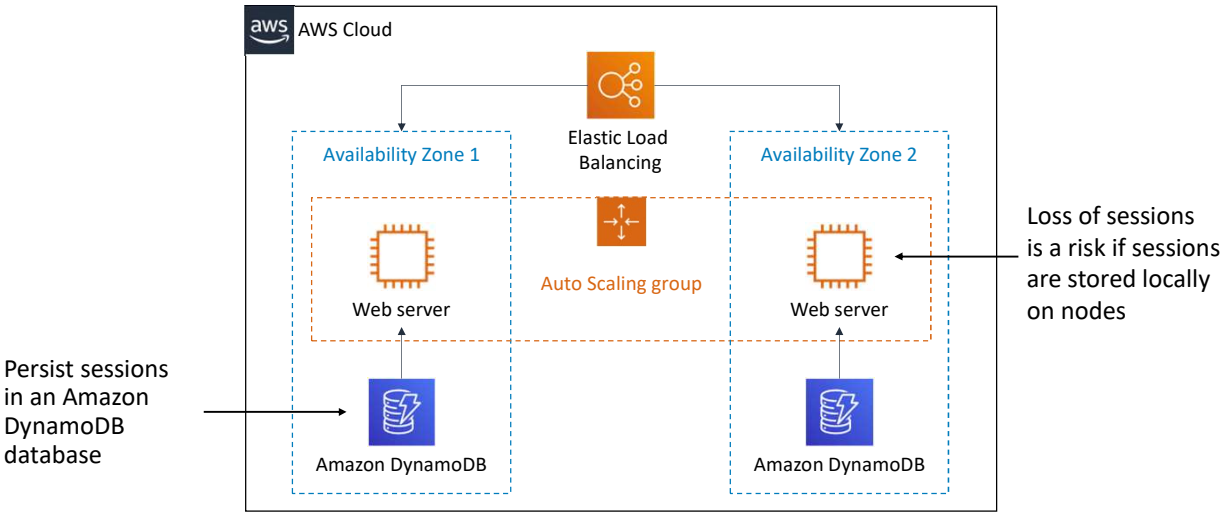
Another disadvantage of sticky sessions is that they can limit your application's scalability. With sticky sessions, the load balancer is unable to truly balance the load each time that it receives a request from a client. Sticky sessions force the load balancer to send all the requests to the original server where the session state was created. If that server is heavily loaded, then receiving so many requests can lead to unequal load across servers and affect user response time.

Instead of sticky sessions: Persist sessions inside a distributed cache



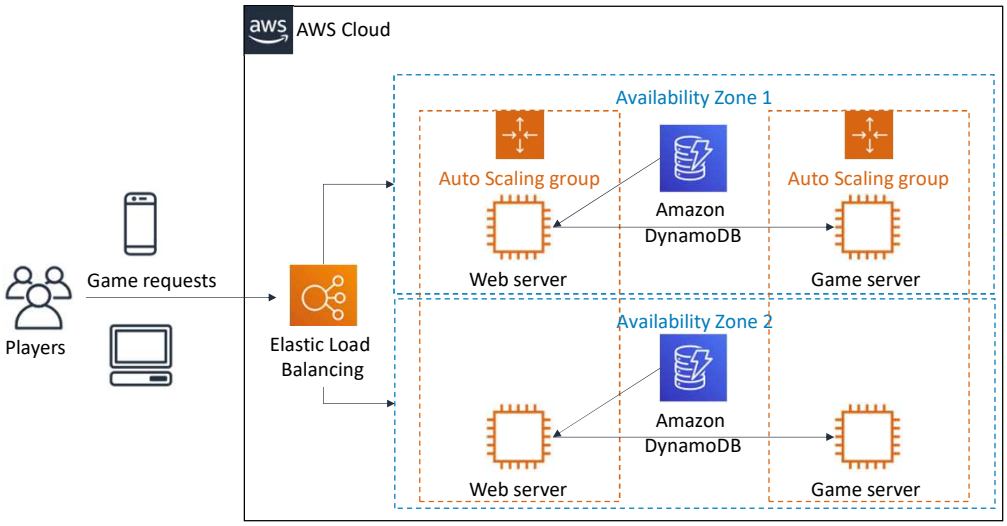
Instead of using sticky sessions, you can designate a layer in your architecture that can store sessions in a scalable and robust manner outside the instance. One option is to persist session data in a distributed cache, as this architecture diagram shows. It makes sense to implement this architecture in a dynamic environment when the number of web servers changes to accommodate load, and you don't want to risk losing sessions.

Instead of sticky sessions: Persist sessions inside a DynamoDB table



Another option is to persist session data in an Amazon DynamoDB database, as this diagram shows. With Amazon DynamoDB, you can set your scaling policy and have DynamoDB scale capacity up and down as necessary.

Example: Storing session states for an online gaming application



This architecture might support an online gaming application, where fast session retrieval is imperative. Game data is continuously updated as a player collects items, defeats enemies, receives gold, unlocks levels, and completes achievements. Each session event must be written to your database layer so that it isn't lost. Game makers store session history and other time-oriented data in DynamoDB for fast lookup by player, date, and time.

Each DynamoDB database table is associated with a throughput capacity. You can specify 1,000 writes per second, and DynamoDB scales the database in the background. As your needs change, you can update the capacity—and Amazon DynamoDB re-allocates resources as needed. This elasticity helps game developers: if your game becomes popular, you might suddenly scale from a few thousand players to millions of players. You can quickly and easily scale back down if needed.

DynamoDB maintains predictable, low-latency performance at any scale, which is crucial if your game grows to millions of latency-sensitive customers. You won't spend any time tuning the performance of DynamoDB.

To see a similar gaming architecture that includes DynamoDB, see this AWS Big Data Blog post.

Section 4 key takeaways

- Sessions are used to manage user authentication and store user data while the user interacts with the application.
- You can manage sessions with sticky sessions, which is a feature of Elastic Load Balancing load balancers. Sticky sessions route requests to the specific server that's managing the user's session.
- You can also manage sessions by persisting session data outside the web server instance—for example, in a distributed cache or DynamoDB table.

Some key takeaways from this section of the module include:

- Sessions are used to manage user authentication and store user data while the user interacts with the application.
- You can manage sessions with sticky sessions, which is a feature of Elastic Load Balancing load balancers. Sticky sessions route requests to the specific server that is managing the user's session.
- You can also manage sessions by persisting session data outside the web server instance—for example, in a distributed cache or DynamoDB table.

Module 11: Caching Content

Section 5: Caching databases

aws academy

Introducing Section 5: Caching databases.

When should you cache your database?

You are concerned about response times for your customer.

You have a high volume of requests that are inundating your database.

You would like to reduce your database costs.

As you learned earlier in this module, time-consuming database queries and complex queries can create bottlenecks in applications.

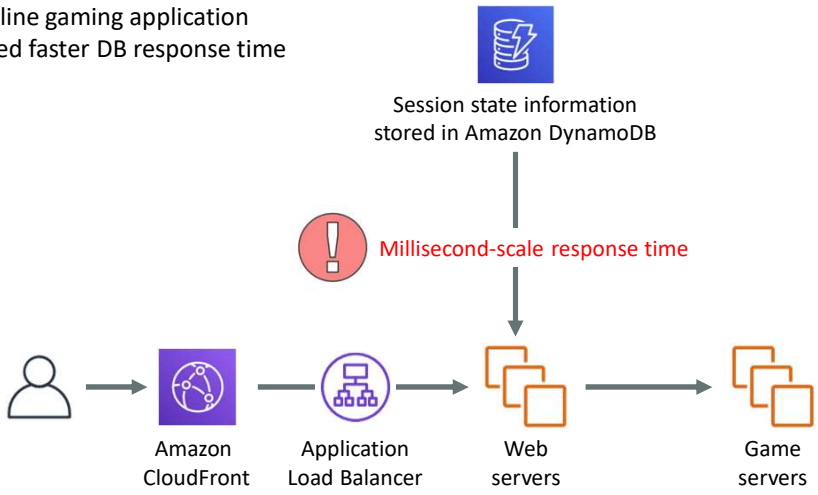
You should consider caching your database when you:

- Are concerned about response times for your customer. You might have latency-sensitive workloads that you want to speed up. Caching can help you increase throughput and reduce data retrieval latency, thus improving the performance of your application.
- Have a high volume of requests that inundate your database. You might have a large amount of traffic, and thus not getting the throughput that you need for that workload. Putting a caching layer next to your database can increase your throughput and help you achieve higher performance.
- Would like to reduce your database costs. Whether data is distributed in a disk-based NoSQL database or vertically scaled up in a relational database, scaling for high reads can be costly. A number of database read replicas might be necessary to match what a single in-memory cache node can deliver in terms of requests per second.

A database cache supplements your primary database by removing unnecessary pressure on it, typically in the form of frequently accessed read data. The cache itself can be in a number of areas, including your database, application, or as a standalone layer.

Using DynamoDB for state information

Use case: Online gaming application
Problem: Need faster DB response time



Consider a simplified version of the architecture for an online gaming application, where you are storing session state information in DynamoDB. In some cases, you might find that a millisecond-scale response time isn't fast enough for your application. Database caching can help with this problem.

Amazon DynamoDB Accelerator

Fully managed, highly available, in-memory cache for DynamoDB

- Extreme performance (microsecond-scale response time)
- Highly scalable
- Fully managed
- Integrated with DynamoDB
- Flexible
- Secure

Amazon DynamoDB Accelerator (DAX) is a fully managed, highly available, in-memory cache for DynamoDB. It delivers up to a performance improvement of up to 10 times—from milliseconds to microseconds—even at millions of requests per second. DAX does all the heavy lifting required to add in-memory acceleration to your DynamoDB tables. You are not required to manage cache invalidation, data population, or cluster management.

DAX provides the following benefits:

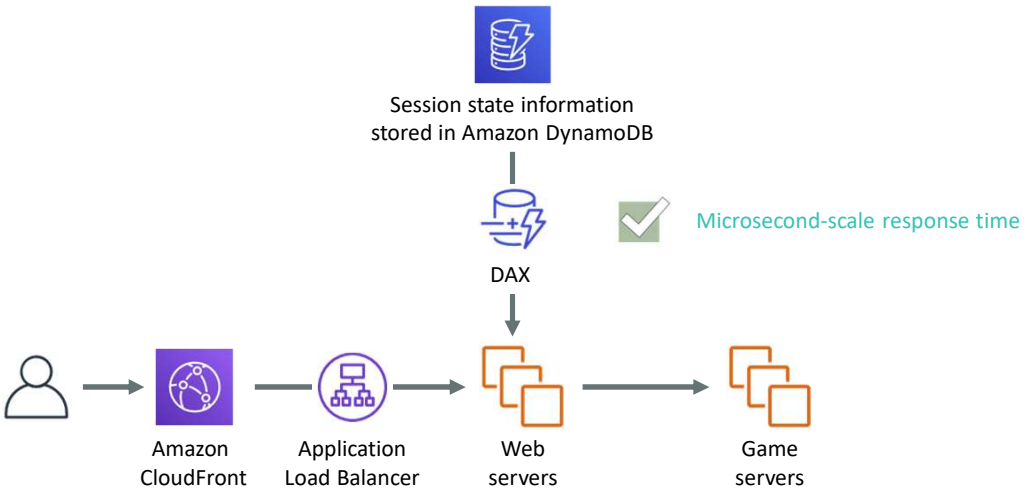
- **Extreme performance** – DynamoDB offers consistent single-digit millisecond latency. When DynamoDB and DAX are used together, you can achieve response times in microseconds for millions of requests per second for read-heavy workloads.
- **Highly scalable** – DAX has on-demand scaling. You can start with a three-node DAX cluster and add capacity as necessary, up to a 10-node cluster.
- **Fully managed** – Like DynamoDB, DAX is fully managed. DAX takes care of management tasks including provisioning, setup and configuration, software patching, and replicating data over nodes during scaling operations. DAX automates common administrative tasks such as failure detection, failure recovery, and software patching.
- **Integrated with DynamoDB** – DAX is API-compatible with DynamoDB, and it's not necessary to make any functional application code changes. Provision a DAX cluster, and use the DAX client software development kit (SDK) to point existing DynamoDB API calls at the DAX cluster. DAX handles the rest.

- **Flexible** – You can provision one DAX cluster for multiple DynamoDB tables, multiple DAX clusters for a single DynamoDB table, or a combination of both.
- **Secure** – DAX fully integrates with AWS services to enhance security. You can use AWS Identity and Access Management (IAM) to assign unique security credentials to each user and control each user's access to services and resources. Amazon CloudWatch enables you to gain system-wide visibility into resource utilization, application performance, and operational health. Integration with AWS CloudTrail enables you to easily log and audit changes to your cluster configuration. DAX supports Amazon Virtual Private Cloud (Amazon VPC) for secure and easy access from your existing applications. Tagging provides you more visibility to help you manage your DAX clusters.

The retrieval of cached data reduces the read load on existing DynamoDB tables. As a result, it might reduce provisioned read capacity and lower overall operational costs.

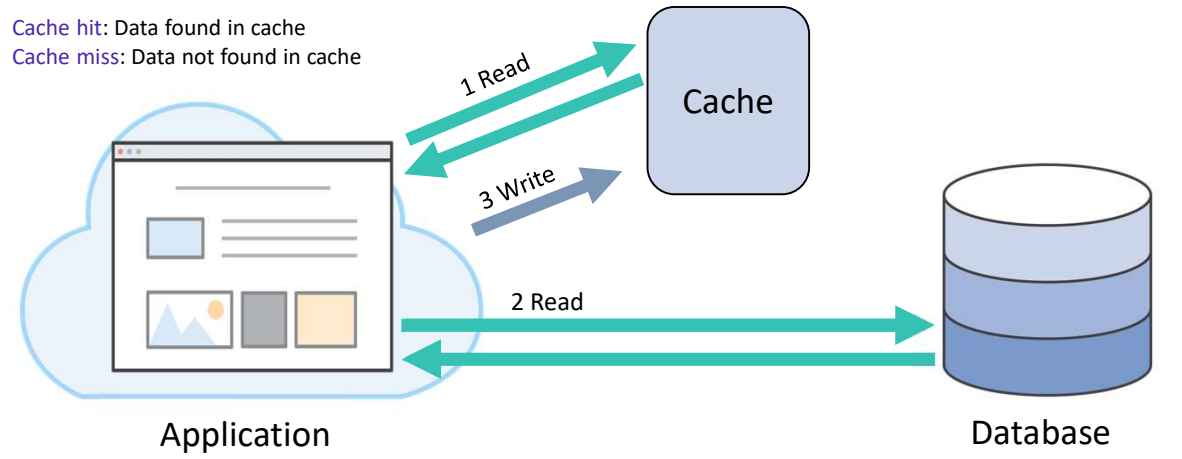
For more information about DAX, see this AWS Database Blog post.

Using DynamoDB with DAX to accelerate response time



In the gaming example, the acceleration that you get by adding DAX to your architecture comes without needing to make any major changes in the game code. In this way, it simplifies deployment into your architecture. The one thing you must do is re-initialize your DynamoDB client with a new endpoint that points to DAX. It isn't necessary to make any changes to the rest of the code. DAX handles cache invalidation and data population without your intervention. This cache can help speed responsiveness when you run events that might cause a spike in players. An example of such an event is a seasonal downloadable content (DLC) offering or a new patch release.

Remote or side caches



DAX is a transparent cache. Another way to implement a database cache deployment is to use a remote or side cache. A side cache isn't directly connected to the database—instead, it's used adjacently to the database. Side caches are typically built on key-value NoSQL stores such as Redis or Memcached. They provide hundreds of thousands of requests, up to a million requests per second per cache node.

Side caches are typically used for read-heavy workloads. They work as follows:

1. For a given key-value pair, an application first tries to read the data from the cache. If the cache contains the data (called a *cache hit*), the value is returned.
2. If the intended key-value pair is not found in the cache (called a *cache miss*), the application fetches the data from the underlying database.
3. It's important that the data is present when the application needs it again. To ensure that it is, the key-value pair that's obtained from the database is then written to the cache.

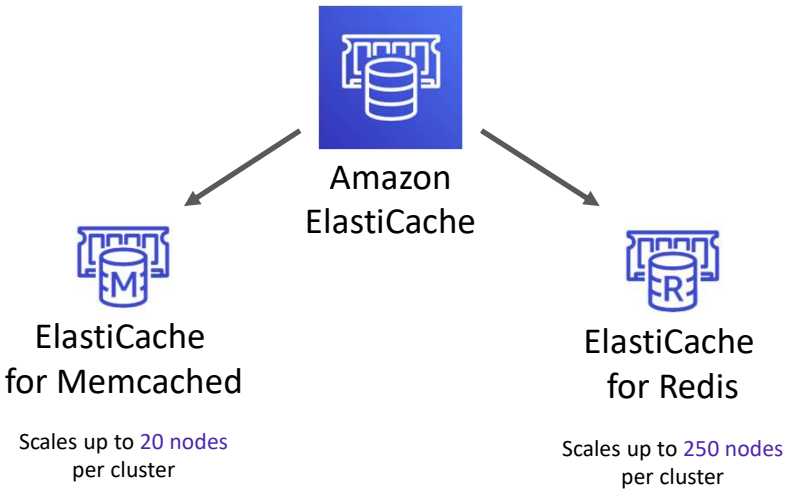
Amazon ElastiCache

ElastiCache provides web applications with an in-memory data store in the cloud.

- Works an in-memory data store and cache
- Offers high performance
- Is fully managed
- Is scalable
- Supports Redis and Memcached

Amazon ElastiCache is a side cache that works as an in-memory data store to support the most demanding applications, which require sub-millisecond response times. With ElastiCache, you don't need to perform management tasks such as hardware provisioning, software patching, setup, configuration, monitoring, failure recovery, and backups. ElastiCache continuously monitors your clusters to keep your workloads up and running so that you can focus on higher-value application development. Amazon ElastiCache can scale out, scale in, and scale up to meet fluctuating application demands. Write and memory scaling is supported with sharding. Replicas provide read scaling. ElastiCache supports two open-source in-memory databases: Redis and Memcached.

Redis and Memcached



ElastiCache for Memcached can scale up to 20 nodes per cluster. In contrast, ElastiCache for Redis can scale up to 250 nodes for increased data access performance. ElastiCache supports Amazon VPC, which enables you to isolate your cluster to the IP ranges that you choose for your nodes.

ElastiCache runs on the same highly reliable infrastructure that other AWS services use. ElastiCache for Redis provides high availability through Multi-AZ deployments with automatic failover. For Memcached workloads, data is partitioned across all nodes in the cluster. Thus, you can scale out to better handle more data when demand grows.

Memcached versus Redis comparison

Feature	Memcached	Redis
Sub-millisecond latency	Yes	Yes
Ability to scale horizontally for writes and storage	Yes	No
Multi-threaded performance	Yes	No
Advanced data structures	No	Yes
Sorting and ranking datasets	No	Yes
Publish/subscribe messaging	No	Yes
Multi-AZ deployments with automatic failover	No	Yes
Persistence	No	Yes

The Memcached and Redis engines are simple caches that can be used to offload DB burden. Each engine provides certain advantages. This table compares some key features between Memcached and Redis.

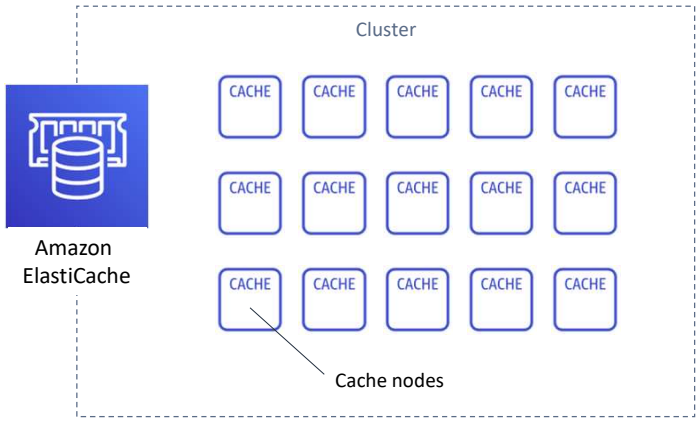
- **Sub-millisecond latency** – Both engines offer sub-millisecond response times. By storing data in memory, they can read data more quickly than disk-based databases.
- **Ability to scale horizontally** – Memcached enables you to scale out and in, adding and removing nodes as demand on your system increases and decreases.
- **Multi-threaded performance** – Because Memcached is multithreaded, it can use multiple processing cores, which means that you can handle more operations by scaling up compute capacity.
- **Advanced data structures** – Redis supports complex data types, such as strings, hashes, lists, sets, sorted sets, and bitmaps.
- **Sorting or ranking datasets** – You can use Redis to sort or rank in-memory datasets. For example, you can use Redis sorted sets to implement a game leaderboard that keeps a list of players that is sorted by their rank.

- **Publish/subscribe messaging** – Redis supports publish/subscribe messaging with pattern matching, which you can use for high-performance chat rooms, real-time comment streams, social media feeds, and server intercommunication.

- **Multi-AZ deployments with automatic failover** – ElastiCache for Redis provides high availability through Multi-AZ deployments with automatic failover, in case your primary node fails.

- **Persistence** – Redis enables you to persist your key store. By contrast, the Memcached engine does not support persistence. For example, perhaps a node fails and is replaced with a new, empty node; or you might terminate a node or scale one down. In this case, you lose the data that's stored in cache memory.

ElastiCache components

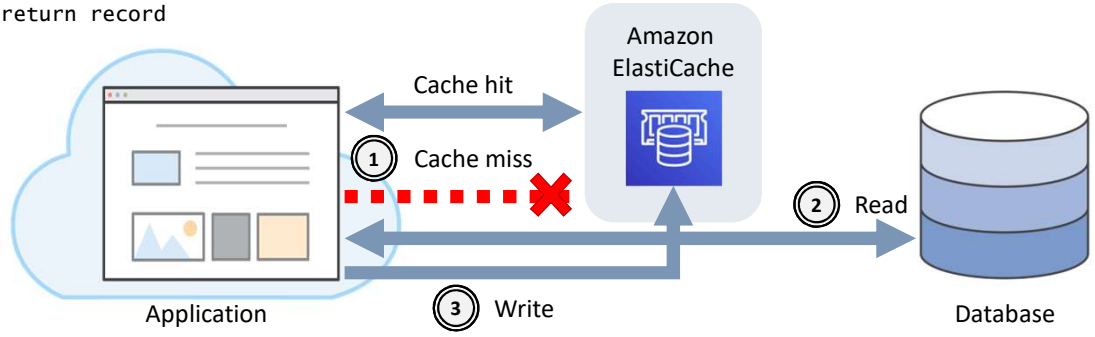


- A node is the smallest block of an ElastiCache deployment
- Each node has its own DNS name and port
- A cluster is a logical grouping of one or more nodes

A cache *node* is the smallest building block of an ElastiCache deployment. It is a fixed-size chunk of secure, network-attached RAM. Each node runs the engine that was chosen when the cluster or replication group was created or last modified. Each node has its own DNS name and port. It can exist in isolation from other nodes, or in a grouping with other nodes, which is also known as a *cluster*.

Caching strategies: Lazy loading

```
def get_user(user_id):
    # Check the cache
    record = cache.get(user_id)
    if record is None:
        # Run a DB query
        record = db.query("select * from users where id = ?", user_id)
        # Populate the cache
        cache.set(user_id, record)
    return record
```



You can employ two caching strategies with ElastiCache.

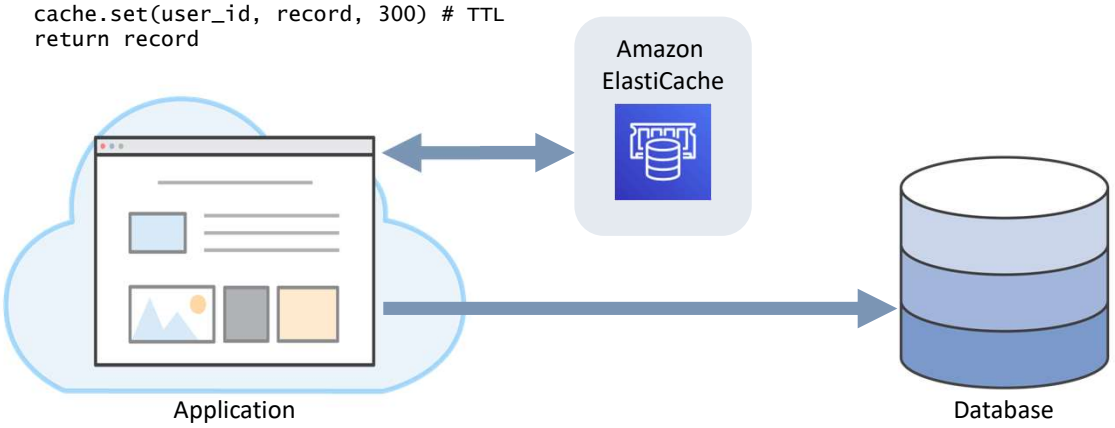
The first strategy, *lazy loading*, is a caching strategy that loads data into the cache only when necessary. In this case, when your application requests data, it first makes the request to the ElastiCache cache. If the data exists in the cache and is current, a cache hit occurs and ElastiCache returns the data to your application. Otherwise (in the case of a cache miss), your application requests the data from your data store, which returns the data to your application. Your application then writes the data to the cache so that it can be retrieved more quickly the next time that it's requested.

Use lazy loading when you have data that will be read often, but written infrequently. For example, in a typical web or mobile application, a user's profile rarely changes, but it is accessed throughout the application. A person might update his or her profile only a few times per year. However, the profile might be accessed dozens or hundreds of times a day, depending on the user.

The advantage of lazy loading is that only requested data is cached. Because most data is never requested, lazy loading avoids filling up the cache with unnecessary data. However, a cache miss incurs a penalty. Each cache miss results in three trips, which can cause a noticeable delay in data getting to the application. Also, if data is written to the cache only when a cache miss occurs, data in the cache can become stale. Lazy loading does not provide updates to the cache when data is changed in the database.

Caching strategies: Write-through

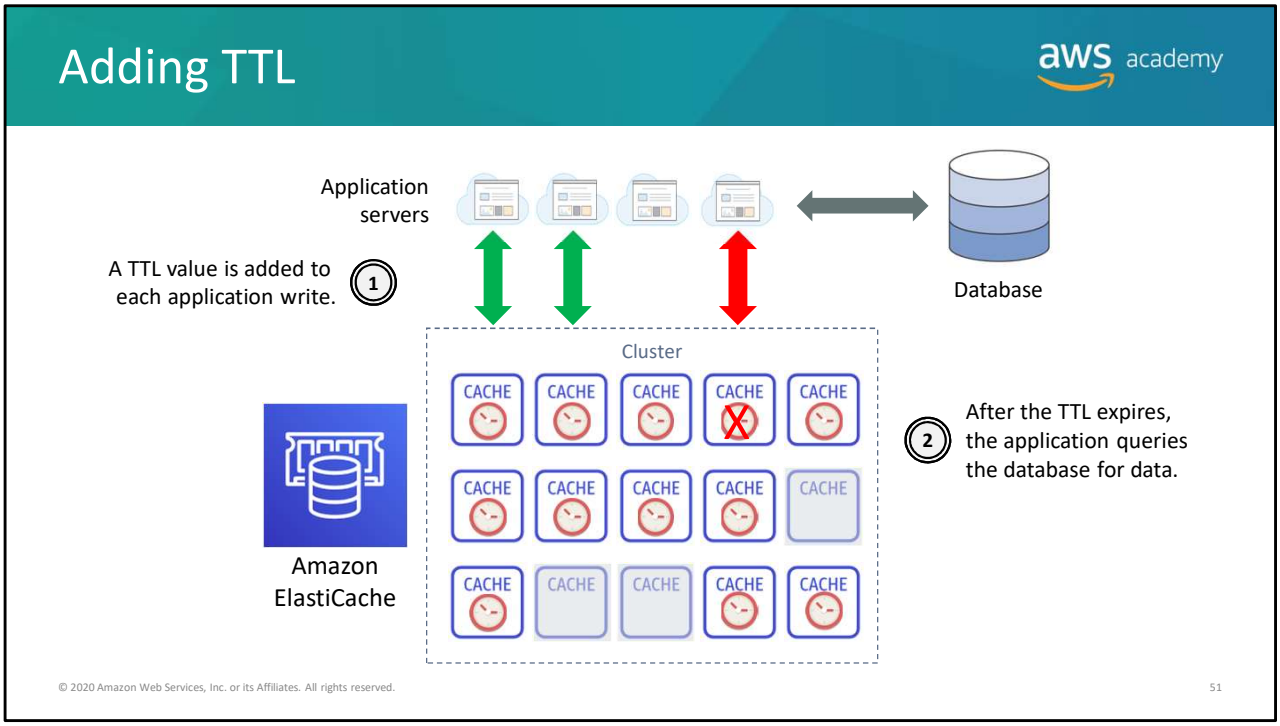
```
def save_user(user_id, values):
    # Save to DB
    record = db.query("update users…where id = ?", user_id, values)
    # Push into cache
    cache.set(user_id, record, 300) # TTL
    return record
```



The *write-through* strategy is the second caching strategy. It adds data or updates data in the cache when data is written to the database. Use a write-through caching strategy when you have data that must be updated in real time. This approach is proactive: you can avoid unnecessary cache misses when you have data that you know is going to be accessed. A good example for this situation is any type of aggregate, such as a top-100-game leaderboard, the top-10-most-popular news stories, or recommendations. Because this data is typically updated through a specific piece of application or background job code, it's straightforward to update the cache along with it.

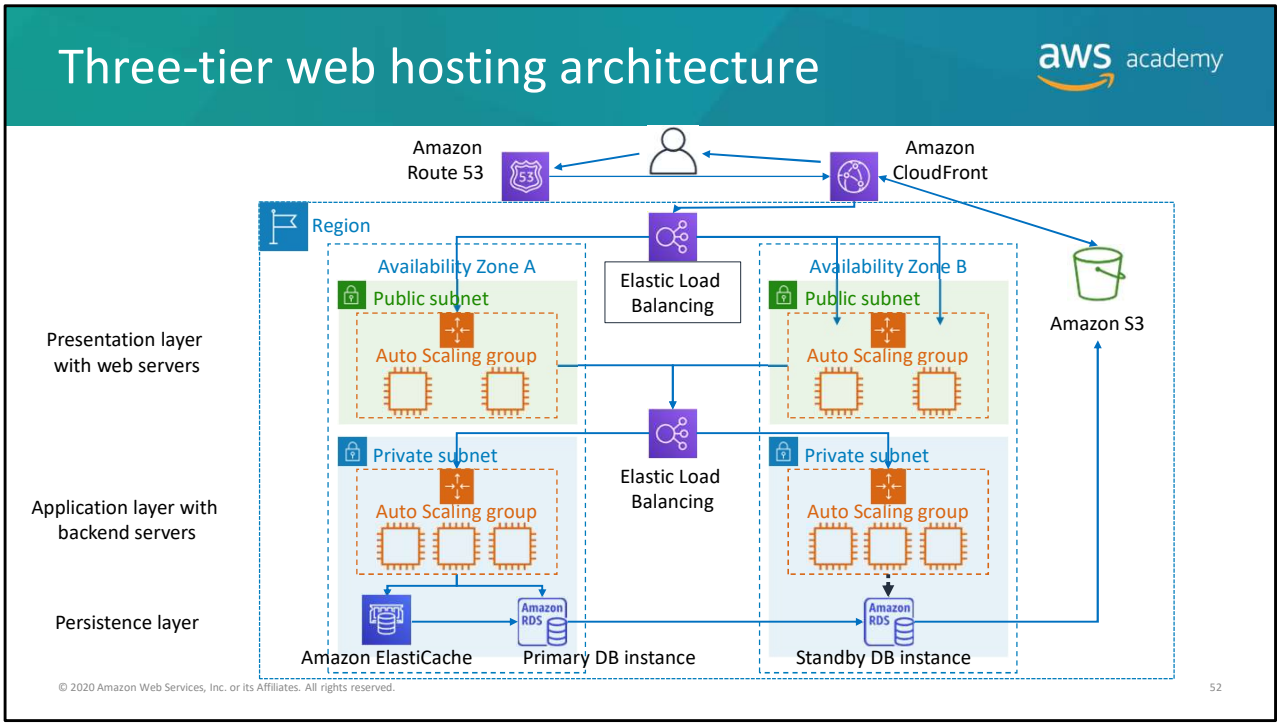
The advantage of this approach is that it increases the likelihood that your application will find that value in the cache when it looks for it. The disadvantage is that you are potentially caching data that you don't need, so this approach can cause added costs.

In practice, both approaches are used together, so it's important for you to understand the frequency of data change and use the appropriate TTLs.



Lazy loading permits stale data. Write-through ensures that data is always fresh, but this approach might populate the cache with unnecessary data.

By adding a TTL value to each write, you enjoy the advantages of each strategy and avoid cluttering the cache with data. TTL is an integer value or key that specifies the number of seconds or milliseconds, depending on the in-memory engine, until the key expires. When an application attempts to read an expired key, it's treated as though the data isn't found in the cache. As a result, the database is queried and the cache is updated. This way, the data doesn't get too stale, and values in the cache are occasionally refreshed from the database.



One scenario where you might want to use Amazon ElastiCache is in a traditional, three-tier web hosting architecture. In this scenario, you want to run a public web application while you still maintain private backend servers in a private subnet. You can create a public subnet for your web servers that have access to the internet. At the same time, you can place your backend infrastructure in a private subnet with no internet access. The database tier of your backend infrastructure might include Amazon Relational Database Service (Amazon RDS) DB instances and an ElastiCache cluster that provides the in-memory layer.

In this web hosting architecture diagram:

- *Amazon Route 53* enables you to map your zone apex (such as *example.com*) DNS name to your load balancer DNS name.
- *Amazon CloudFront* provides edge caching for high-volume content.
- A load balancer spreads traffic across web servers in Auto Scaling groups in the presentation layer.
- Another load balancer spreads traffic across backend application servers in the Auto Scaling groups that are in the application layer.
- *Amazon ElastiCache* provides an in-memory data cache for the application, which removes load from the database tier.

Section 5 key takeaways

- A database cache supplements your primary database by removing unnecessary pressure on it, typically in the form of frequently accessed read data
- DAX is a fully managed, highly available, in-memory cache for DynamoDB that delivers a performance improvement of up to 10 times—from milliseconds to microseconds
- Amazon ElastiCache is a side cache that works as an in-memory data store to support the most demanding applications that require sub-millisecond response times

Some key takeaways from this section of the module include:

- A database cache supplements your primary database by removing unnecessary pressure on it, typically in the form of frequently accessed read data
- DAX is a fully managed, highly available, in-memory cache for DynamoDB that delivers a performance improvement of up to 10 times—from milliseconds to microseconds
- Amazon ElastiCache is a side cache that works as an in-memory data store to support the most demanding applications requiring submillisecond response times

Module 11: Caching Content

Module wrap-up

aws academy

It’s now time to review the module and wrap up with a knowledge check and discussion of a practice certification exam question.

Module summary

In summary, in this module, you learned how to:

- Identify how caching content can improve application performance and reduce latency
- Create architectures that use Amazon CloudFront to cache content
- Identify how to design architectures that use edge locations for distribution and distributed denial of service (DDoS) protection
- Recognize how session management relates to caching
- Describe how to design architectures that use Amazon ElastiCache

In summary, in this module, you learned how to:

- Identify how caching content can improve application performance and reduce latency
- Create architectures that use Amazon CloudFront to cache content
- Identify how to design architectures that use edge locations for distribution and distributed denial of service (DDoS) protection
- Recognize how session management relates to caching
- Describe how to design architectures that use Amazon ElastiCache

Complete the knowledge check

It is now time to complete the knowledge check for this module.

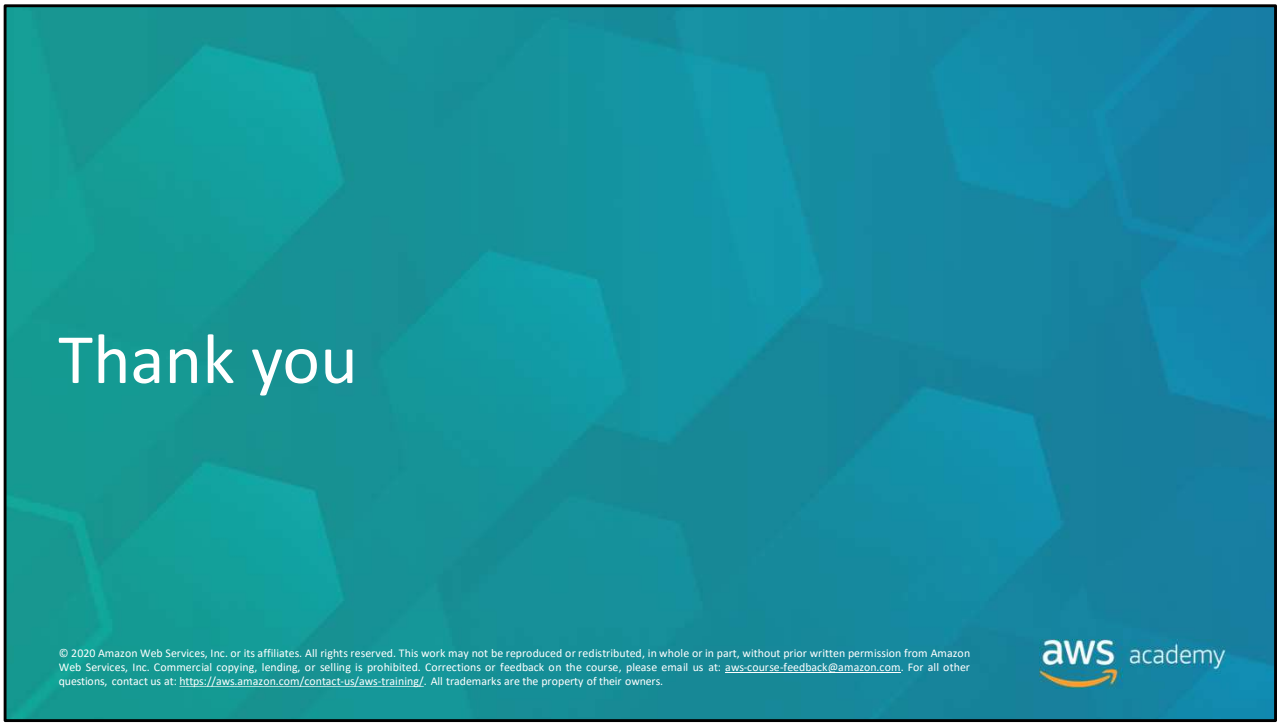
Sample exam question

A company is developing a highly available web application that uses stateless web servers. Which services are suitable for storing session state data? (Select TWO.)

A. Amazon CloudWatch
B. Amazon DynamoDB
C. Elastic Load Balancing
D. Amazon ElastiCache
E. AWS Storage Gateway

Look at the answer choices and rule them out based on the keywords that were previously highlighted.

The correct answers are B (Amazon DynamoDB) and D (Amazon ElastiCache): Both DynamoDB and ElastiCache provide high-performance storage of key-value pairs. CloudWatch and Elastic Load Balancing are not storage services. AWS Storage Gateway is a storage service, but it's a hybrid storage service that enables on-premises applications to use cloud storage.



Thank you for completing this module.