MODULE 12 · STUDY NOTES

Caching Content

How caching cuts latency and offloads origins — Amazon CloudFront edge caching as a CDN, Amazon ElastiCache for database caching, and Well-Architected caching strategies

Speed shapes whether an application succeeds, and caching is the architect's main lever for it. A cache is a high-speed storage layer that keeps a subset of data closer to whoever needs it, so requests are answered faster and the slower, more expensive origin is spared. This module covers the two AWS caching services and where each belongs. Amazon CloudFront is a content delivery network (CDN) that performs static, or edge, caching — it keeps copies of web content in a global network of edge locations near your users and delivers them with low latency, while also strengthening security against DDoS attacks. Amazon ElastiCache performs database, or data, caching — a managed, in-memory, key-value store that sits between your application and its database to cut latency and offload read pressure. Threaded through both is the central challenge of caching: data goes stale by nature, so an architect must choose deliberately how long to cache (TTL, versioning, headers, invalidation for CloudFront) and how to keep the cache in sync (lazy loading or write-through for ElastiCache), always balancing performance, cost, and data freshness against the specific business need. The module closes by mapping these choices onto the AWS Well-Architected Framework.

LEARNING OBJECTIVES

- Identify how caching content can improve application performance and reduce latency
- Identify how to use Amazon CloudFront to deliver content by using edge locations
- Create architectures that use CloudFront to cache content
- Describe how to use Amazon ElastiCache for database caching
- Use the AWS Well-Architected Framework principles when designing caching strategies

12.1 Overview of caching

One way to make a system faster is to make its data available more quickly. Many systems have some pieces of data that are accessed far more often than others; temporarily storing that data on a technology faster than its natural location permits quicker access. A **cache** is a high-speed data storage layer that stores a *subset* of data so that future requests for that data are served up faster than is possible by reaching the data's primary storage location. Unlike a database — which stores data in a complete and **durable** form — a cache stores data **transiently**. Caching lets you efficiently reuse previously retrieved or computed data; it is an optimization strategy that both reduces cost and enhances performance, and for the end user it simply means faster performance.

WORKED EXAMPLE – THE TOOL SHED ANALOGY

Imagine a hardware store several miles away. Going there every time you need a common item costs real effort. Instead, you keep the supplies you use regularly in a **tool shed** near your house, so reaching them takes far less time — and you still visit the store occasionally to refresh your stock. The tool shed is the cache: a nearby place holding the things you need often. In AWS, that nearby location takes different forms — **edge locations** (CloudFront) and an **in-memory database** (ElastiCache) — both of which this module explores.

WHAT SHOULD YOU CACHE?

- **Static and frequently accessed data** — content that rarely changes (HTML, CSS, JavaScript, images, video) and is requested often, such as a social-media profile. Don't cache data if caching yields no speed or cost advantage.
- **Results of computationally intensive calculations** — recommendation engines and high-performance computing simulations benefit from an in-memory data layer acting as a cache.
- **Results of time-consuming, frequently used, or complex database queries** — queries that join multiple tables are slow and expensive; if data requires a slow, costly query, it is a candidate for caching.

BENEFITS OF CACHING	CHALLENGES OF CACHING
why add a cache	*what it costs you*
▪ Reduces response latency for users ▪ Reduces cost ▪ Alleviates load on the origin (data source) ▪ Provides predictable performance ▪ Improves availability	▪ Requires additional engineering — a new component adds complexity ▪ Someone must decide how to cache data and handle stale data, based on the business use case ▪ Needs a clear strategy for when to update or invalidate cached data to avoid incorrect behavior

Because a cache adds an extra component to the infrastructure, make sure there is a **legitimate need** justified by cost, latency, or availability improvements, and plan to operate it with the same rigor as the rest of your fleet. Be deliberate and empirical about **cache size, expiration policy, and eviction policy**. The two types of caching in this module map to two services, summarized next.

TABLE 12.1 Types of caching and AWS caching services

TYPE	AWS SERVICE	HOW IT WORKS
Static caching (edge caching)	Amazon CloudFront	A CDN uses its global network of edge locations to deliver a cached copy of web content (videos, webpages, images) from the edge nearest the requester
Database caching (data caching)	Amazon ElastiCache	An in-memory cache acts as an adjacent data-access layer to any relational or NoSQL database, loaded via techniques such as lazy loading and write-through

COMMON PITFALL

A cache is not a database. A database stores data in a complete, durable form and is the system of record; a cache stores only a **subset** of data, **transiently**, purely to speed retrieval. Caching improves application speed and decreases database cost — it does not replace the underlying data store.

12.2 Caching using CloudFront

The delivery challenge: because of the global, complex nature of the internet, traffic between an application and its users travels large physical distances in both directions. When a user is far from the server, a large file such as a video or image takes a long time to load. **The CDN solution:** a **content delivery network** is a globally distributed system of caching servers that introduces intermediary servers between the client and the application. A CDN caches copies of commonly requested static files and delivers a **local copy** from the nearby cache edge or **point of presence (PoP)** that

provides the fastest delivery — decreasing traffic to the web server, reducing bandwidth consumption, and improving the user experience.

WHAT AN EDGE CACHE CAN AND CANNOT CACHE

- **Can cache (static):** web objects (HTML documents, CSS style sheets, JavaScript files), image files, and video files.
- **Cannot cache:** dynamically generated content and user-generated data.
- **But can still deliver:** you can configure CloudFront to deliver dynamic content from a **custom origin** — for example, an EC2 instance or a web server — it just isn't cached.
- **Security:** configure CloudFront to require viewers to use **HTTPS**, and to use HTTPS to fetch from the origin, so connections are encrypted in both directions.

CloudFront is a CDN service built for high performance, security, and developer convenience. It delivers content across the globe securely with low latency and high transfer speeds by routing each user request through the **AWS backbone network** to the edge location that can best serve the content — dramatically reducing the number of networks a request must traverse and increasing reliability, because copies of your content are cached in edge locations around the world. CloudFront also improves resiliency against **distributed denial of service (DDoS)** attacks by working with **AWS Shield** and **AWS WAF**, and it delivers over HTTPS using the latest TLS version.

TABLE 12.2 CloudFront global edge network components

ATTRIBUTE	EDGE LOCATIONS	REGIONAL EDGE CACHES
Quantity & proximity	More numerous and closer to users	Fewer and farther from users
Cache size	Smaller caches	Larger caches — objects remain cached longer
Content role	Serve popular content quickly; may drop objects as they become less popular	Hold less-popular content (user-generated media, ecommerce assets, news)
Position	Serve viewers directly	Sit between your origin and the edge locations; reduce origin fetches

To deliver content with lower latency, CloudFront uses a global network of **more than 550 edge locations** and **13 Regional edge caches** across many countries (as of October 2023). Edge locations are connected to AWS Regions through the redundant AWS network backbone, improving origin fetches and dynamic content acceleration.

HOW CACHING WORKS IN CLOUDFRONT – THE REQUEST FLOW

- **1.** A user makes a request to access content (for example, the `cat.jpg` image file).
- **2.** DNS routes the request to the edge location that can best serve it — typically the nearest, with the shortest latency. CloudFront checks the cache.
- **3. Cache hit:** if the content is in the cache, CloudFront delivers it immediately and the journey ends.
- **4. Cache miss:** if not cached, CloudFront forwards the request to the **origin** you identified (for example, an S3 bucket) as the source of the definitive content.
- **5.** The origin sends the object back to the edge location.
- **6.** CloudFront forwards the file to the user and **adds it to the cache** for the next request. The **TTL** setting determines how long the edge caches content before requesting it again from the origin.

HOW TO CONFIGURE A CLOUDFRONT DISTRIBUTION

- **Specify an origin location** — where CloudFront gets your files. For HTTP, the origin is an **S3 bucket** or an HTTP server (a **custom origin**) that can run on EC2 or on-premises.
- **Configure the distribution** — tell CloudFront which origins to use, and specify details such as request logging and whether the distribution is enabled on creation.
- **CloudFront becomes available** — it assigns a domain name to your new distribution.
- **CloudFront distributes the configuration** — it sends the distribution's configuration (but *not* the content) to all edge locations, where copies of objects are later cached.

Controlling how long content is cached. If you update content but still see the old version, CloudFront is likely still serving cached content. CloudFront serves cached content from an edge location until it expires; after expiry, the next request causes CloudFront to fetch from the origin and re-cache. Edge locations automatically expire content after the maximum TTL elapses (by default, 24 hours). The best practice is to understand and use four approaches **together**.

TABLE 12.3 Four ways to control CloudFront caching duration

METHOD	HOW IT WORKS	TRADE-OFF OR NOTE
Set a maximum TTL	CloudFront may expire content anytime between the minimum and maximum TTL; default is 24 hours. Lower the maximum to expire content faster	More frequent origin requests, since caches repopulate more often
Implement content versioning	Embed a version identifier (timestamp, sequential number) in file names; new names make CloudFront fetch immediately	Sidesteps CloudFront expiration behavior altogether
Specify Cache-Control headers	Keep the minimum TTL at 0 so CloudFront honors any `Cache-Control: max-age` header set per file	Gives precise control over expiration for individual files
Use invalidation requests	Force CloudFront to expire content on demand	Not immediate (takes minutes); use sparingly and only for individual objects

Video streaming with CloudFront. CloudFront can deliver streaming video, but you must first use an **encoder** to format and package the content. Examples include **AWS Elemental MediaConvert** and **Amazon Elastic Transcoder** (which transcodes media into versions that play on phones, tablets, and PCs). After converting into output formats, you host the converted content in an **S3 bucket** (the origin) and use CloudFront to deliver the segment files worldwide. Packaging creates **segments** (static files of audio, video, and captions) and **manifest files** (which describe which segments to play and in what order). Package formats include **DASH (MPEG-DASH)**, **Apple HLS**, **Microsoft Smooth Streaming**, and **CMAF**. The module's guided lab, *Streaming Dynamic Content Using Amazon CloudFront*, builds exactly this pattern with a CloudFront distribution and an Elastic Transcoder pipeline.

DDOS MITIGATION – CLOUDFRONT WITH ROUTE 53, WAF, AND SHIELD

- **The challenge:** publicly accessible applications and APIs are exposed to threats (OWASP Top 10 vulnerabilities, SQL injection, automated requests, HTTP floods) that harm availability and security. A DDoS attack floods the target from many sources to make it unavailable.
- **Amazon Route 53 + CloudFront:** always-on monitoring, anomaly detection, and mitigation of common infrastructure DDoS attacks are built into both; DNS requests and application traffic routed through CloudFront are inspected inline.
- **AWS WAF:** CloudFront can create a **web access control list (web ACL)** with rules that analyze incoming requests and block common web threats before they reach your servers.
- **AWS Shield:** DDoS mitigations continually inspect traffic and allow only valid web traffic through to CloudFront — automatic protection against common vectors such as UDP reflection attacks. Distributing content through CloudFront protects your systems as a side benefit.

COMMON PITFALL

Delivering dynamic content is not the same as caching it. CloudFront happily **delivers** dynamic and user-generated content from a custom origin, but an edge cache only **caches static** content. When a question asks what an edge cache stores, the answer is images, videos, and web objects — never dynamically generated or user-generated data.

12.3 Caching using ElastiCache

Time-consuming and complex database queries create bottlenecks in applications. A **database cache** supplements your primary database by removing unnecessary pressure on it, typically for frequently accessed **read** data. The cache can live inside your database or application, or as a standalone layer. ElastiCache provides that layer as a managed service.

WHEN SHOULD YOU CACHE YOUR DATABASE?

- **You are concerned about response times** — latency-sensitive workloads you want to speed up. Caching increases throughput and reduces retrieval latency.
- **A high volume of requests is overburdening your database** — a caching layer next to the database raises throughput and helps you reach the performance you need.
- **You want to reduce database costs** — scaling for high reads is costly; a single in-memory cache node can deliver the requests-per-second that would otherwise require many database read replicas.

Amazon ElastiCache is a fully managed, key-value, in-memory data store with **sub-millisecond** latency that sits between an application and an origin data store. It decreases access latency, eases the load on databases and applications, provides a high-performance and cost-effective in-memory cache, and is fully compatible with the open-source **Redis** and **Memcached** engines.

BENEFITS OF ELASTICACHE

- **Fully managed** — provisions resources, installs software, and automates failure detection, recovery, and patching; provides detailed monitoring metrics.
- **Scalable** — can scale out, in, and up to meet fluctuating application demand.
- **Extreme performance** — an in-memory store delivering sub-millisecond response times for the most demanding applications.
- **Cost-effective** — pay the price of an expensive query (such as a multi-table join) **one time**, then retrieve the cached result many times.
- **Memcached or Redis compatible** — keep using the same code, drivers, and tools; deploy, run, and scale open-source-compatible in-memory data stores.

TABLE 12.4 Choosing an ElastiCache engine

CAPABILITY	ELASTICACHE FOR MEMCACHED	ELASTICACHE FOR REDIS
Data model	Simplest model, low maintenance	Complex data types: strings, hashes, lists, sets, sorted sets, bitmaps
Threading & scaling	Multithreaded — large nodes with multiple cores/threads; **Auto Discovery**; up to 20 nodes	Scale up to 250 nodes; nodes grouped into **shards** (1–6 nodes each)
Persistence	None — data is lost if a node is replaced	**Persists** the key store
Sort / rank	Not supported	Sort or rank datasets (e.g., a game leaderboard)
High availability	Not available	**Multi-AZ** with automatic failover
Messaging	Not supported	**Publish/subscribe** (chat, real-time feeds)

ELASTICACHE COMPONENTS

- **Node** — the smallest building block: a fixed-size chunk of secure, network-attached RAM running the chosen engine, with its own DNS name and port. You can scale nodes up or down to a different instance type.
- **Cluster** — a logical grouping of one or more nodes. Your application connects to a node or cluster through a unique address called an **endpoint**.
- **Memcached partitioning** — data is partitioned across all nodes in the cluster (up to 20), so you scale out to handle more data.
- **Redis shard** — a Redis cluster is a logical grouping of one or more **shards**; a shard is a grouping of 1–6 related nodes. Redis clusters scale up to 250 nodes with Multi-AZ automatic failover.

Time to live (TTL). A TTL value is added to each write, specifying the number of seconds or milliseconds (depending on the engine) until the key expires. When an application reads an **expired** key, it is treated as though the data were not in the cache — so the database is queried and the cache is updated. This keeps data from getting too stale and refreshes cached values from the database occasionally. Because data goes stale by nature, you pair TTL with a strategy for keeping the cache in sync.

TABLE 12.5 Strategies for handling stale data

FEATURE	LAZY LOADING	WRITE-THROUGH
Caching pattern	Updates the cache after the data is requested (loads on a cache miss)	Updates the cache immediately after updating the primary database
Advantage	The cache contains only data the application actually requests	The cache stays up to date with the database — data is most likely found in the cache
Disadvantage	Requires a programmatic strategy to keep the cache current; permits stale data	Increases cost from storing in-memory data you might never use
Best fit	Data read often but written infrequently (e.g., a user profile)	Data that must be updated in real time

Lazy loading in detail: the application first tries to read from the cache. A **cache hit** returns the data and ends the journey; a **cache miss** forwards the request to the primary database, which returns the data to the user and adds it to the cache for next time. Its advantage is that only requested data is cached, so the cache never fills with data no one asks for — but each cache miss incurs a penalty of **three trips**, and data can become stale because lazy loading does not update the cache when the database changes. **Write-through** in detail: whenever the application changes data, it writes to the database and **immediately** updates the cache. This is proactive — the value is very likely to be in the cache when the application looks, and cached data is always current — but you may cache data you never need, adding cost. Choose based on the impact of stale data: high impact favors write-through for freshness; low impact makes lazy loading sufficient. The frequency of change in the underlying data drives the performance and cost trade-off.

> COMMON PITFALL
>
> Persistence is a Redis-only feature. Memcached does **not** persist data — if a node fails and is replaced with an empty node, or you delete or scale one down, the data in that cache memory is lost. If a scenario requires persistence, complex data types, ranking/leaderboards, Multi-AZ failover, or pub/sub, the answer is **Redis**; if it just needs a simple, multithreaded cache, choose **Memcached**.

12.4 Applying AWS Well-Architected Framework principles to caching

The AWS Well-Architected Framework has **six pillars**, each with best practices and questions to consider when architecting solutions. This section highlights a few pillar best practices most relevant to caching. As you connect the pillars to caching, keep the cloud architect's core concerns in view: **when** to cache content to improve performance and optimize cost, **how** to deal with stale content to balance cost against data freshness, and **why** to incorporate a CDN and an in-memory caching strategy into your designs.

TABLE 12.6 Well-Architected best practices relevant to caching

PILLAR	FOCUS AREA	BEST-PRACTICE GUIDANCE
Performance efficiency	Data management	Implement data access patterns that use caching; implement strategies to improve query performance in the data store
Reliability	Foundations — plan your network topology	Use highly available network connectivity for your workload's public endpoints
Cost optimization	Cost-effective resources — plan for data transfer	Perform data-transfer modeling; select components and implement services to optimize and reduce data-transfer cost

Performance efficiency (data management). Storing data in a cache improves read latency, read throughput, user experience, and efficiency while reducing cost. To *implement data access patterns that use caching*: first identify databases, APIs, and network services that would benefit (services with heavy read workloads, a high read-to-write ratio, or that are expensive to scale); pick the caching strategy that fits the access pattern; configure a cache-invalidation strategy such as a **TTL** that balances freshness against backend pressure; and **monitor the cache hit rate with a goal of 80 percent or higher** — a lower rate may indicate an insufficient cache size or an access pattern that does not benefit from caching. To *improve query performance in the data store*, a distributed caching solution improves latency and reduces the number of database I/O operations.

Reliability (plan your network topology). It is critical to plan, build, and operate highly available network connectivity for your public endpoints: if a workload becomes unreachable, customers see it as down even when it is running. One best practice is to *use highly available network connectivity for public endpoints* via **CDNs**. CloudFront distributes content with low latency and high transfer rates through edge locations near users, and it shifts content load away from your servers to edge locations and Regional edge caches — improving reachability and availability, and (as covered earlier) protecting against DDoS attacks.

Cost optimization (plan for data transfer). Using the right services, resources, and configurations is key to savings. *Perform data-transfer modeling* to find the lowest-cost point for current requirements and understand where transfer occurs, what it costs, and its benefit. *Select components to optimize data-transfer cost* — look for alternative architectures, such as using **CDNs to locate data closer to users**. *Implement services to reduce data-transfer costs* — use edge locations or CDNs to deliver content, or build caching layers in front of application servers and databases. CloudFront caches data at edge locations worldwide, reducing load on your resources and the effort of serving large global audiences with minimal latency (a security savings bundle can save up to 30 percent on CloudFront usage over time). Controlling how long content is cached — via minimum/maximum TTL, versioning, `Cache-Control` headers, and invalidation — is itself a cost lever.

> **WORKED EXAMPLE – THE MODULE'S SAMPLE EXAM QUESTION**
>
> *Your company hosts an ecommerce site with records in an Amazon RDS database. After a few days, RDS suffers serious performance issues due to website requests. Which option is the most cost-effective way to deal with performance issues due to many read requests?* Isolate the key phrases: **Amazon RDS database**, **cost-effective**, **performance issues due to many read requests** — this points to a caching solution. **Choice A** (enable Multi-AZ for RDS) provides high availability but not performance. **Choice C** (place a load balancer in front of the database) does not affect database performance. **Choice D** (configure CloudFront to cache from the database) is wrong because CloudFront is not used for database caching. **Choice B — configure Amazon ElastiCache to cache frequently used content from the database — is correct**, because it reduces the amount of read traffic sent to the database.

Module summary

- A cache is a high-speed storage layer that transiently holds a subset of data so future requests are served faster; good candidates are static/frequently accessed data, compute-intensive results, and slow or complex query results.
- Caching reduces latency and cost, offloads the origin, and improves availability — but it adds a component, and you must deliberately decide how to cache and how to handle stale data for the business case.
- Two types map to two services: static (edge) caching with Amazon CloudFront, and database (in-memory) caching with Amazon ElastiCache.
- CloudFront is a CDN that serves content from a global network of edge locations (numerous, close, small caches for popular content) and Regional edge caches (fewer, farther, larger caches for less-popular content, between origin and edge).
- CloudFront request flow: DNS routes to the best edge; a cache hit serves immediately, a cache miss fetches from the origin (e.g., an S3 bucket), caches, then delivers. Edge caches only static content but can deliver dynamic content from a custom origin.
- Control CloudFront caching duration with four approaches used together: maximum TTL (24-hour default), content versioning, Cache-Control headers, and invalidation requests (which are not immediate).
- CloudFront also secures delivery — HTTPS/TLS by default, and DDoS resilience via Amazon Route 53, AWS WAF web ACLs, and AWS Shield; it can stream video from S3 after an encoder packages it into segments and manifests.
- ElastiCache is a fully managed, key-value, in-memory store with sub-millisecond latency, compatible with Memcached (simple, multithreaded, Auto Discovery, up to 20 nodes) and Redis (complex data types, persistence, ranking, Multi-AZ failover, pub/sub, up to 250 nodes; nodes group into shards). A node is the smallest block; a cluster groups nodes.
- Keep the ElastiCache cache fresh with TTL plus a strategy — lazy loading (loads on a cache miss, caches only requested data but permits stale data and a three-trip penalty) or write-through (updates the cache on every write, always fresh but costlier). Well-Architected ties caching to performance efficiency (cache hit rate goal 80% +), reliability (highly available public endpoints via CDN), and cost optimization (data-transfer reduction).

Review questions

1. **What is a cache, and how does it differ from a database?**

   Answer: A cache is a high-speed data storage layer that transiently stores a subset of data so future requests are served faster. A database stores data in a complete, durable form as the system of record; a cache is an optimization layer holding only a subset, transiently, to speed retrieval.

2. **Name three kinds of content that are good candidates for caching.**

   Answer: Static and frequently accessed data (HTML, CSS, JavaScript, images, video); results of computationally intensive calculations; and results of time-consuming, frequently used, or complex database queries such as multi-table joins.

3. **What are the two types of caching in this module, and which AWS service provides each?**

   Answer: Static caching (edge caching) via Amazon CloudFront, which serves content from the nearest edge location; and database caching (data caching) via Amazon ElastiCache, which serves content from an in-memory layer next to the database.

4. **How do CloudFront edge locations differ from Regional edge caches?**

   Answer: Edge locations are more numerous, closer to users, and have smaller caches that serve popular content quickly. Regional edge caches are fewer, farther from users, and have larger caches that hold less-popular content longer; they sit between the origin and the edge locations and reduce origin fetches.

5. **In CloudFront, what happens on a cache hit versus a cache miss?**

   Answer: On a cache hit, CloudFront serves the content from the edge location immediately and the journey ends. On a cache miss, CloudFront forwards the request to the origin (for example, an S3 bucket), the origin returns the object to the edge location, and CloudFront delivers it to the user while adding it to the cache for next time.

6. **List the four ways to control how long CloudFront caches content.**

   Answer: Set a maximum TTL (default expiry 24 hours), implement content versioning (new file names fetch immediately), specify Cache-Control headers (per-file control when minimum TTL is 0), and use invalidation requests (force expiry, though not immediately).

7. **When would you choose ElastiCache for Redis over Memcached?**

   Answer: Choose Redis when you need complex data types, persistence of the key store, sorting or ranking of datasets (leaderboards), high availability with Multi-AZ automatic failover, or publish/subscribe messaging. Choose Memcached for the simplest model, multithreaded large nodes, and horizontal scaling with Auto Discovery.

8. **Compare lazy loading and write-through, with one advantage and one disadvantage of each.**

   Answer: Lazy loading updates the cache after data is requested (on a cache miss): advantage — only requested data is cached; disadvantage — permits stale data and pays a cache-miss penalty of three trips. Write-through updates the cache immediately after every database write: advantage — the cache is always current; disadvantage — it can cache data you never use, adding cost.