MODULE 9 · STUDY NOTES

Caching Information for Scalability

Why and where applications cache, in-memory data stores with Amazon ElastiCache, global delivery with Amazon CloudFront and edge functions, and the lazy-loading and write-through strategies

Frank and Martha's café website has a speed problem: the coffee-bean inventory that the application pulls from its database is slow to display, and customers notice. Sofía also wants the whole site to load quickly for users everywhere and to move from HTTP to secure HTTPS. Both goals point to the same technique — **caching**, serving requests from a fast copy of the data instead of returning to the slow original every time. This module builds that out on two fronts. **Amazon ElastiCache** places an in-memory, sub-millisecond key-value store between the application and its database to cut query latency, and **Amazon CloudFront** places a global content delivery network in front of the site so content is served from an edge location near each viewer, over HTTPS. Along the way it covers where caching belongs in an application, how to decide what is worth caching, the fresh-versus-fast trade-off that every TTL sets, the two CloudFront edge-function options (Lambda@Edge and CloudFront Functions), and the two strategies — lazy loading and write-through — for keeping a cache in step with its origin.

LEARNING OBJECTIVES

- Explain when caching is used
- Describe caching with Amazon ElastiCache
- Describe caching with Amazon CloudFront
- Describe and compare Lambda@Edge and CloudFront Functions
- Apply caching strategies

9.1 Introduction: the café requirement

Café employees have gotten occasional customer feedback that the **coffee-bean inventory information takes too long to display** on the café website. Olivia, an AWS consultant and café customer, proposed adding **database caching** to improve the speed of the query that returns that inventory. Separately, Sofía wants to **reduce latency for users globally** and update the site to **support HTTPS** traffic, so she plans to set up a content delivery network (CDN). These two needs — a faster database read and faster, secure global delivery — map directly to the two services this module explores and to its two hands-on labs.

TABLE 9.1 The two labs and what each one solves

LAB	PROBLEM	WHAT YOU BUILD
9.1	Bean-inventory query is slow	Add database caching with **Amazon ElastiCache for Memcached** in front of the Aurora Serverless database
9.2	Site is HTTP-only and slow far from the origin	Configure an **Amazon CloudFront** distribution (with AWS WAF and a CloudFront Function) to cut latency and serve content securely over **HTTPS**

In the completed application, end users reach the café website through **AWS WAF**, which fronts a **CloudFront** distribution that caches the static website content stored in **Amazon S3**; WAF also protects an **API Gateway** endpoint whose Lambda function reads menu items from **DynamoDB**. On the supplier side, an Elastic Beanstalk-deployed container queries an **ElastiCache for Memcached** cluster before falling back to the Aurora Serverless database. Caching, in other words, appears at more than one layer of the same application.

9.2 Caching overview

Modern applications place ever-higher demands on their data sources. As applications moved from monolithic designs toward distributed, microservice-based ones, they went from single-server apps serving hundreds of users to globally distributed apps serving **millions** — all sending requests and expecting immediate responses. Think of Internet of Things (IoT) fleets sending millions of requests, a bidding site where a few hundred **microseconds** change the outcome, or a massive global game whose players constantly interact. Developers must rethink how they store and access data to meet these performance and scale demands, and **caching is one strategy** for doing so.

At the simplest level, **caching is serving requests from a copy of data rather than requesting it from the original source**. The **cache** is that copy and the **origin** is the source; the cache is a high-speed layer that stores data **transiently**, and each item has a **TTL** so it doesn't get too stale. The primary purpose of a cache is to **increase performance of data retrieval by reducing the need to access the origin** — delivering faster responses to the client, reduced load on downstream services, or both. You saw this pattern in Amazon API Gateway: with its cache enabled, API Gateway checks the cache first and returns the response directly if the data is there, avoiding a trip to the backend.

YOU CAN CACHE AT EVERY LAYER OF AN APPLICATION

- **Client side** — accelerate retrieval of web content in the browser with HTTP `Cache-Control` headers.
- **Server side (web content and sessions)** — use Amazon CloudFront, ElastiCache for Redis, ElastiCache for Memcached, and AWS Partner solutions.
- **Application performance and data access** — use ElastiCache for Redis, ElastiCache for Memcached, and AWS Partner solutions.
- **Database query caching** — reduce query latency with ElastiCache for Redis and ElastiCache for Memcached.
- This module focuses on **ElastiCache** and **CloudFront**, but you might use other options at different layers depending on your requirements.

READ-HEAVY WORKLOADS	COMPUTE-INTENSIVE WORKLOADS
caching reduces latency and improves IOPS	*caching manipulates large datasets in real time*
▪ Benefit: reduce latency and improve input/output operations per second (IOPS) ▪ Social networking ▪ Gaming ▪ Media sharing ▪ Question-and-answer (Q&A) portals	▪ Benefit: manipulate large datasets in real time across clusters of machines ▪ Recommendation engines ▪ High performance computing (HPC) simulations ▪ Disk-based stores become a bottleneck at this scale

Not everything belongs in a cache. Based on application requirements, developers choose how often to refresh the cache and which origin data to cache, weighing four questions.

TABLE 9.2 Factors for deciding whether to cache

ASK YOURSELF	WHY IT MATTERS
Is a slow, expensive query required to generate the data?	Databases are slower and pricier than a cache; joins across multiple tables are especially expensive — good caching candidates. Pay the query cost once.
Is the data relatively static and frequently accessed?	Cache benefits static, frequently read data (a social profile). Don't cache rapidly changing or rarely accessed data, or unique results like search pages.
Can the data afford to be stale?	Treat cached data as stale by definition. A delayed stock price with a disclaimer is fine; the same price for a broker executing a trade needs real-time data.
Is caching worth the added complexity and cost?	Weigh benefit against the effort to write caching code, an added attack surface, and cost. Don't optimize too early — find bottlenecks first.

COMMON PITFALL

Don't optimize too early. Wait until the application is complete so you can see where the real bottlenecks are, monitor access patterns and performance in production-like scenarios, and then apply caching where it adds the most benefit. Caching is a trade-off, not a free win: it can add development complexity, a new security vector, and cost.

9.3 Caching with ElastiCache

Amazon ElastiCache is an in-memory, key-value store that sits between your application and the data store it accesses. Its purpose is ultrafast (**sub-millisecond latency**) and inexpensive access to copies of data: it decreases access latency, increases throughput, and eases the load on databases and applications, and it can scale out, in, up, or down to meet demand. Because querying a database is always slower and more expensive than locating a key in a key-value cache — especially for queries that join multiple tables or run intensive calculations — you can cache those results, **pay the price of the query once**, and then retrieve the data many times without rerunning it. ElastiCache is fully compatible with the open-source **Redis** and **Memcached** engines.

WHY ELASTICACHE IS A MANAGED SERVICE

- **Fully managed** — it provisions server resources, installs software, and automates failure detection, recovery, and patching; detailed monitoring metrics let you set thresholds and receive alarms when a node is overloaded.
- **Extreme performance** — an end-to-end optimized stack on customer-dedicated nodes supports the most demanding, sub-millisecond applications.
- **Scalable** — scale out, in, and up to meet fluctuating application demand.

TABLE 9.3 Memcached and Redis, feature by feature

FEATURE	ELASTICACHE FOR MEMCACHED	ELASTICACHE FOR REDIS
Threading	Multithreaded — can use more CPU cores than Redis	Single-threaded model
Maintenance	Low maintenance; simple by design	More features, more to manage
Scaling	Easy horizontal scaling with Auto Discovery of node changes	Shards and replicas (see terminology below)
Data types	Flat strings and objects (HTML, serialized JSON)	Complex types: strings, hashes, lists, sets, sorted sets, bitmaps
Persistence	None — a replacement or scaled-down node loses its data	Persistent — can be used as a primary data store
Availability	No replication	Replication to read replicas; Multi-AZ automatic failover
Messaging	Not available	Publish/subscribe channels
Choose when	You need the simplest model possible	You need persistence, advanced data types, or the above

The decision rule is simple: **base the choice on a use case that justifies Redis.** Choose Memcached if you need the simplest possible model and your cache doesn't need Redis's advanced features; choose Redis if you need data persistence, advanced data types, or any of its other capabilities such as replication, automatic failover, or pub/sub.

KEY TERMS

Node
A fixed-size chunk of secure, network-attached RAM. Each node runs one instance of the chosen engine and version, has its own DNS name and port, and can be scaled up or down to a different instance type.

Cluster
A logical grouping of one or more ElastiCache nodes. Every node in a cluster is the same instance type and runs the same cache engine.

Endpoint
The unique address your application uses to connect to an ElastiCache node or cluster.

Memcached partitioning
With Memcached, data is partitioned across nodes in a cluster — up to 20 nodes — and there is no replication.

TABLE 9.4 Redis terminology (console names, with API/CLI names)

CONSOLE TERM	API / CLI TERM	DEFINITION
Redis shard	Node group	A grouping of 1–6 related nodes. With more than one node it supports replication: one read/write primary and the rest read-only replicas.
Redis cluster	Replication group	A logical grouping of one or more shards. Without cluster mode: one shard, no partitioning. With cluster mode: up to 500 shards, data partitioned across them.

ElastiCache integrates with services such as Amazon EC2, Amazon RDS, and Amazon DynamoDB, and with deployment tools such as AWS CloudFormation, AWS Elastic Beanstalk, and AWS OpsWorks — so you focus on application logic rather than infrastructure. In a typical architecture, Elastic Load Balancing spreads traffic across EC2 application instances; the application reads from the cache and, only when the data is missing or expired, reads from the database (Amazon RDS, DynamoDB, or another). When the database serves the same data repeatedly, the application writes it into the cache so the next read is fast.

> **WORKED EXAMPLE – THE CACHE-MISS PENALTY AND TTL**
>
> Your code handles a small loop: request data from the cache; on a **cache hit** (found and not expired) return it; on a **cache miss** (absent or the key's TTL has expired) request it from the origin, return it, and update the cache. Notice that a miss means **three trips** — request the cache, query the database, update the cache — which noticeably delays the response. If an application cannot tolerate that penalty for any request, one strategy is to load **all** data into the cache in advance rather than a subset. You must also choose a **TTL** — an integer number of seconds until a key expires. A short TTL causes more misses; a long TTL risks returning stale data. A real page often caches many kinds of data at once (profile, top news, recommendations, comments) with different update methods, so a good baseline practice is to **apply a TTL to every cache key**, then test with production-like data and keep monitoring.

> **SAMPLE EXAM QUESTION, WORKED**
>
> *Which requirement is a reason to choose Memcached over Redis for your ElastiCache configuration?* The key phrases are **requirement**, **Memcached over Redis**, and **ElastiCache**. The correct answer is **"You need the simplest model possible."** Rule out the distractors: both Memcached and Redis are in-memory caches, so "you need an in-memory cache" doesn't distinguish them; **data replication** and **complex data types** are Redis features that Memcached does not provide. Memcached wins only when you want the simplest model.

9.4 Caching with CloudFront

Serving data globally introduces a problem that in-memory caching alone doesn't solve. When someone uses your application, the request is routed through many networks to reach your **origin server** — the source that stores the original, definitive versions of your objects (webpages, images, media files). The **number of network hops** and the **distance** the request travels significantly affect performance, and network latency varies by geographic location. That is where a content delivery network comes in.

A **content delivery network (CDN)** is a globally distributed system of caching servers. It caches copies of commonly requested static files — HTML, CSS, JavaScript, images — that the origin server hosts, and delivers a **local** copy from the cache edge, or **Point of Presence (POP)**, that provides the fastest delivery. A CDN can also deliver **dynamic** content that is unique to the requester and not cacheable, improving performance because the POP maintains low-latency, secure connections close to the requester and can proxy content back to the origin efficiently.

Amazon CloudFront is a fast CDN service that securely delivers data, videos, applications, and APIs to customers globally with low latency and high transfer speeds. It is optimized for performance and scale, has built-in security (it integrates with **AWS Shield** for DDoS protection), is deeply integrated with AWS (use **Amazon S3**, ELB, and EC2 as origins), and is highly customizable (it integrates with **Lambda@Edge** to run code closer to users).

HOW CLOUDFRONT WORKS

- A user requests one or more files (for example, an image and an HTML file).
- **DNS routes** the request to the CloudFront edge location (POP) that can best serve it — typically the nearest, lowest-latency POP.
- CloudFront **checks its cache**: if the files are there it returns them; if not, it forwards the request to the appropriate origin for that file type (S3 for images, an HTTP server for HTML).
- The origin sends the files back to the edge location.
- As the first byte arrives, CloudFront **forwards the files to the user and adds them to the edge cache** for the next request.
- **Regional edge caches** sit between POPs and origins to hold content that isn't popular enough to stay at a POP — helpful for user-generated media and content that fades over time.

TABLE 9.5 CloudFront distribution types

TYPE	USED FOR	VALID ORIGINS
Web content	Static and dynamic content over HTTP/HTTPS (.html, .css, .js, images)	HTTP server or S3 bucket
Video on demand	Video watched anytime; must be encoded to support streaming (HLS, Smooth Streaming)	HTTP server or S3 bucket
Live event	Streaming live events; video encoded and compressed for viewing devices	AWS Elemental MediaStore or MediaPackage

The **cache key** is the unique identifier for every object in the cache, and it determines whether a viewer request is a **hit**. By default the cache key is just the **CloudFront distribution domain name plus the URL path** of the requested object — nothing else from the request. So two requests for `/content/stories/example-story.html` produce the **same** cache key and a cache hit even if their query strings, `User-Agent` and `Referer` headers, and `session_id` cookies all differ. A higher **cache hit ratio** means better performance, so include only the **minimum necessary** values in the cache key. You change what goes into the cache key with a **cache policy** — add a value only when it actually changes the response the origin returns, or you will cache duplicate objects.

TABLE 9.6 Cache policy settings categories

CATEGORY	WHAT IT CONTROLS
Policy information	The policy name (used to attach it to a cache behavior) and a comment
TTL settings	Minimum, Maximum, and Default TTL — how long objects stay valid before CloudFront checks the origin; work together with `Cache-Control` and `Expires` headers
Cache key settings	Which viewer-request values join the cache key — URL query strings, HTTP headers, cookies, and compression support; these are also sent on origin requests

Each file expires from a CloudFront edge cache after **24 hours by default**. After a file expires, the next request prompts CloudFront to check the origin: if the cache already has the latest version the origin returns **`304 Not Modified`**; otherwise it returns **`200 OK`** with the latest version. You can change the default duration two ways — set **Minimum/Maximum/Default TTL in a cache policy** (which affects all files matching a path pattern), or add a **`Cache-Control` max-age (or `s-maxage`) or `Expires` header** to the origin response (which affects an individual file). The `Cache-`

`Control` max-age directive is recommended over `Expires`, and if both are set CloudFront uses only max-age. When CloudFront fronts an S3 origin, control expiration in **CloudFront** rather than on the S3 objects, and use Origin Identity Access (OIA) so objects are only ever exposed through CloudFront — just avoid setting conflicting behaviors in both places.

BONUS: AMAZON S3 TRANSFER ACCELERATION

- A **bucket-level** feature that uses CloudFront's edge locations to speed transfers between a client and an S3 bucket over long distances, routing data to S3 over an optimized network path.
- Good when customers upload to one central bucket worldwide, you move gigabytes-to-terabytes across continents, or you can't use all your bandwidth uploading to S3.
- It costs extra and isn't always faster — the **farther** the client is from the S3 Region, the greater the improvement; a nearby client may see no gain or even slower transfers. Test with representative access patterns.

You can also **customize how a distribution processes requests and responses** by attaching an **edge function** — code that runs close to viewers to minimize latency, with no servers to manage. It can manipulate requests and responses, do basic authentication and authorization, or generate HTTP responses at the edge. CloudFront offers two ways to write edge functions.

TABLE 9.7 Choosing between Lambda@Edge and CloudFront Functions

DIMENSION	LAMBDA@EDGE	CLOUDFRONT FUNCTIONS
Use cases	Complex functions and full application logic	Lightweight, short-running functions
Language	Node.js or Python runtime	JavaScript
Event sources	Viewer request/response and origin request/response	Viewer request/response only
Built and tested in CloudFront	No — an extension of AWS Lambda	Yes — a native CloudFront feature
Scale / limits	Greater duration, memory capacity, and function size	Sub-ms duration, 2 MB memory, 10 KB size

WHEN EACH EDGE-FUNCTION OPTION FITS

- **Lambda@Edge** — functions that take milliseconds or more, need adjustable memory or CPU, depend on third-party libraries (including the AWS SDKs), need network access to external services, or need file-system or HTTP-body access. It runs at a **Regional edge cache**; you publish to one Region and CloudFront replicates it worldwide.
- **CloudFront Functions** — cache key normalization, header manipulation, header/URL redirects, and request authorization. Sub-millisecond startup, scales to millions of requests per second, and costs about **one-sixth** of Lambda@Edge.

WORKED EXAMPLE – TWO EDGE FUNCTIONS

A **Lambda@Edge** function can change the origin domain name based on the `CloudFront-Viewer-Country` header — if the viewer is in GB, DE, or IE it rewrites the request origin to `eu.example.com` — reducing latency and providing data sovereignty (to do this you configure the distribution to cache on that header). A **CloudFront Functions** example, attached to the **viewer request** event, returns a `302` redirect from the distribution's domain `d111111abcdef8.cloudfront.net` to `https://aws.amazon.com/cloudfront/` and adds a custom response header — exactly the kind of lightweight redirect CloudFront Functions is built for.

COMMON PITFALL

Match the edge-function option to the work. Lambda@Edge gives you more options and capacity — origin events, larger memory, third-party libraries, and full application logic — but CloudFront Functions is the faster, cheaper choice for tiny, latency-sensitive tasks on viewer request/response, and it is the only one you can build and test entirely inside CloudFront.

9.5 Caching strategies

Caching has two primary strategies — **lazy loading** and **write-through** — that differ in *when* the cache is updated relative to the database. The right choice depends on how the data is read and written; the module uses ElastiCache to illustrate both.

LAZY LOADING	WRITE-THROUGH
cache updated only when necessary (hit or miss)	*cache updated whenever data is written to the DB*
▪ Use when data is **read often but written infrequently** — e.g., a user profile	▪ Use when data **must be updated in real time** — e.g., a top-100 leaderboard or top-10 news stories
▪ Advantage: only requested data is cached — no wasted space	▪ Advantage: **no stale data** — the cache is always current
▪ Advantage: node failures are **non-fatal**; an empty replacement repopulates on misses (with temporarily higher latency)	▪ Disadvantage: **write penalty** — two trips (DB and cache) on every write
▪ Disadvantage: the **cache-miss penalty**	▪ Disadvantage: **missing data** on a new/ replacement node until it is written
▪ Disadvantage: **stale data**, because a database write doesn't update the cache	▪ Disadvantage: **unused data** in the cache and **cache churn**

With **lazy loading**, only requested data is cached. The application requests data; on a miss it reads from the database and updates the cache, so subsequent requests are served fast. Because only touched data is stored, a failed node is not fatal — a new empty node simply repopulates as requests miss and query the database. In pseudocode, a `get_user` function calls `cache.get(user_id)`; if the result is empty it runs a database query and then calls `cache.set(user_id, record)` before returning it. The costs are the miss penalty and potential staleness.

With **write-through**, you update the cache whenever you update the database, so the cached data is never stale. In pseudocode, a `save_user` function writes the record to the database and then calls `cache.set(user_id, record)`. The trade-offs: every write pays a **write penalty** of two trips (though users tolerate latency on updates better than on reads); a newly created or replacement node is **missing data** until records are written again (you might use lazy loading to repopulate it); the cluster can fill with **unused data** that is never read; and repeatedly updated records cause **cache churn**.

CHOOSING A STRATEGY

- Match the strategy to the **use case** for your data.
- **Lazy loading** for read-heavy, write-light data that can tolerate some staleness — it keeps the cache lean and survives node failures.
- **Write-through** for data that must be current in real time — it eliminates staleness at the cost of extra writes and churn.
- The two can be **combined** — for example, use lazy loading to repopulate a write-through cache after a node is replaced.

COMMON PITFALL

There is no single "best" strategy. Lazy loading risks stale data but wastes little space and shrugs off node failures; write-through never serves stale data but pays a write penalty and can fill the cache with unused, churning data. Choose — or combine — based on the read/write pattern and freshness needs of the specific data.

Module summary

- Caching serves requests from a fast, transient copy (the cache) instead of the origin, to increase retrieval performance — faster client responses, lower origin load, or both; every cached item carries a TTL so it doesn't get too stale. You can cache at the client, server, application, and database layers.
- Cache data that is slow/expensive to query, relatively static and frequently accessed, and tolerant of staleness — when the benefit outweighs the added complexity, security surface, and cost. Don't optimize too early: find real bottlenecks first.
- Amazon ElastiCache is a fully managed, in-memory, sub-millisecond key-value store that sits between the application and its data store, running the open-source Memcached and Redis engines.
- Memcached is multithreaded, low-maintenance, and the simplest model, but has no persistence and no replication (data partitions across up to 20 nodes). Redis adds complex data types, persistence, replication/read replicas, Multi-AZ automatic failover, and pub/sub — choose it when you need any of those.
- ElastiCache building blocks: a node is a fixed-size chunk of network-attached RAM, a cluster is a grouping of nodes, and an endpoint is the connection address; Redis adds shards (node groups of 1–6 nodes) and clusters (replication groups, up to 500 shards with cluster mode). A cache miss costs three trips.
- A CDN serves content from globally distributed edge locations (POPs) near the viewer. Amazon CloudFront is a fast, secure CDN (AWS Shield, S3/ELB/EC2 origins) that serves web, video-on-demand, or live-event distributions and returns files from the nearest edge location.
- The CloudFront cache key (default: distribution domain name + URL path) decides hits vs. misses; widen it only through a cache policy. Files expire after 24 hours by default (304 Not Modified vs. 200 OK on revalidation), tunable by cache-policy TTLs (per path pattern) or Cache-Control/Expires headers (per file).
- Edge functions customize request/response processing near viewers: Lambda@Edge (Node.js/Python, viewer and origin events, complex logic, larger limits) vs. CloudFront Functions (JavaScript, viewer events only, sub-ms, 2 MB/10 KB, built entirely in CloudFront, ~1/6 the cost).

- Two caching strategies: lazy loading updates the cache only on a miss (read-often/write-rarely; risks staleness) and write-through updates it on every database write (real-time freshness; write penalty, missing data on new nodes, unused data, churn). Match the strategy to the data.

Review questions

1. **Define a cache and an origin, and state the primary purpose of caching.**

   Answer: A cache is a high-speed storage layer that holds a transient copy of data; the origin is the authoritative source. The primary purpose of caching is to increase data-retrieval performance by reducing the need to access the origin — yielding faster responses, lower origin load, or both.

2. **You have to decide whether a piece of data is worth caching. What four questions do you ask?**

   Answer: Is a slow, expensive query required to generate it? Is it relatively static and frequently accessed? Can it afford to be stale? Is caching worth the added development complexity, security surface, and cost?

3. **What is Amazon ElastiCache, and where does it sit in an application?**

   Answer: A fully managed, in-memory, key-value data store with sub-millisecond latency that sits between the application and the data store it accesses, decreasing access latency, increasing throughput, and easing database and application load; it runs the Memcached and Redis engines.

4. **Name three capabilities Redis has that Memcached lacks, and the one situation where Memcached is preferred.**

   Answer: Redis adds persistence, complex data types, and replication/high availability with automatic failover (plus pub/sub). Memcached is preferred when you need the simplest possible model and don't need those features.

5. **Distinguish a node, a cluster, and an endpoint in ElastiCache.**

   Answer: A node is a fixed-size chunk of secure, network-attached RAM running one engine instance; a cluster is a logical grouping of one or more nodes (all the same instance type and engine); an endpoint is the unique address the application uses to connect to a node or cluster.

6. **By default, what makes up a CloudFront cache key, and how do you get a higher cache hit ratio?**

   Answer: By default the cache key is the distribution domain name plus the URL path of the requested object — nothing else. To raise the hit ratio, include only the minimum necessary values in the cache key, adding query strings, headers, or cookies through a cache policy only when they change the origin's response.

7. **How long does a CloudFront file stay cached by default, and what are the two ways to change that duration?**

   Answer: 24 hours by default. Change it in CloudFront by setting Minimum/Maximum/Default TTL in a cache policy (applies to all files matching a path pattern), or in the origin response header with a Cache-Control max-age (or Expires) header (applies to an individual file); Cache-Control max-age is recommended and wins if both are set.

8. **Compare lazy loading and write-through, and give a use case for each.**

   Answer: Lazy loading updates the cache only on a cache miss — best for data read often but written rarely (a user profile); it keeps the cache lean and survives node failures but risks stale data and pays the miss penalty. Write-through updates the cache on every database write — best for real-time data (a leaderboard or top-stories list); it is never stale but adds a write penalty, leaves new nodes missing data, and causes unused data and churn.