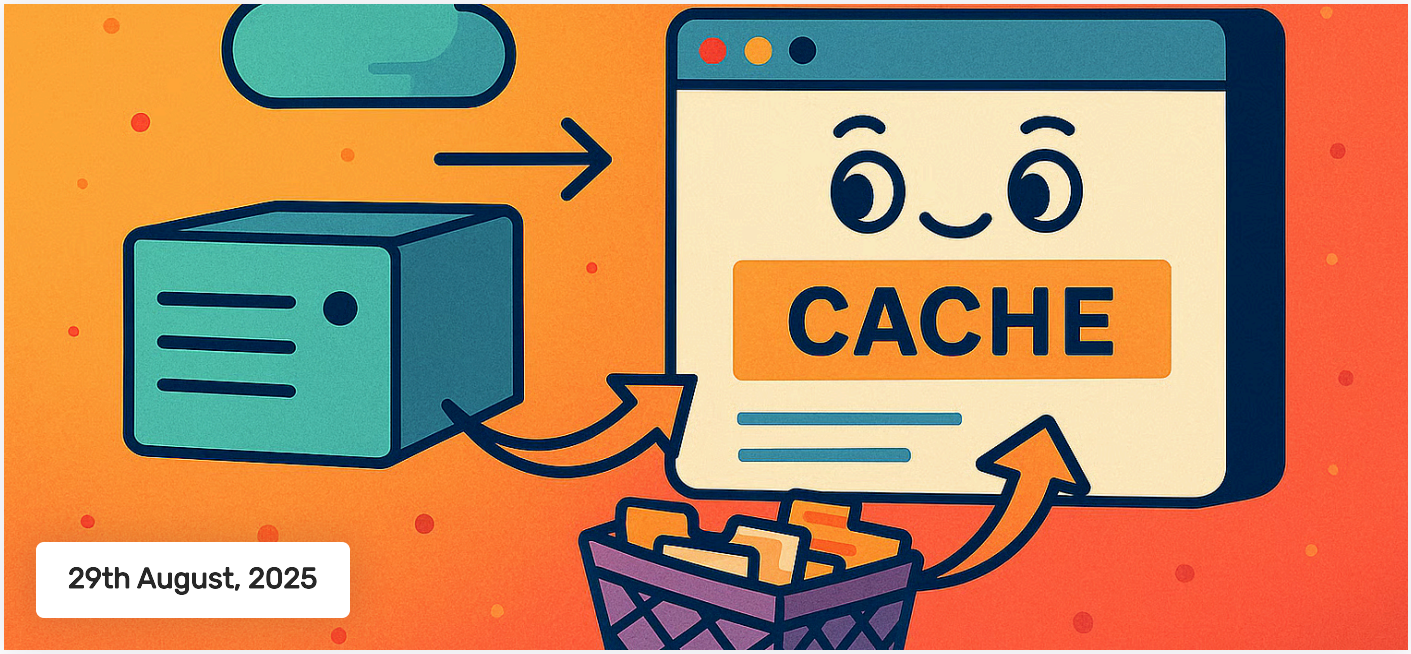
Jono Alderson

A complete guide to HTTP caching



Caching is the invisible backbone of the web. It's what makes sites feel fast, reliable, and affordable to run. Done well, it slashes latency, reduces server load, and allows even fragile infrastructure to withstand sudden spikes in demand. Done poorly – or ignored entirely – it leaves websites slow, fragile, and expensive.

Table of contents

At its core, caching is about reducing unnecessary work. Every time a browser, CDN, or proxy has to ask your server for a resource that hasn't changed, you've wasted time and bandwidth. Every time your server has to rebuild or re-serve identical content, you've added load and cost. Under heavy traffic – whether that's Black Friday, a viral news story, or a DDoS attack – those mistakes compound until the whole stack buckles.

And yet, despite being so fundamental, caching is one of the most misunderstood aspects of web performance. Many developers:

- Confuse `no-cache` with "*don't cache*," when it actually means "*store, but revalidate*".
- Reach for `no-store` as a "safe" default, unintentionally disabling caching entirely.
- Misunderstand how `Expires` interacts with `Cache-Control: max-age`.
- Fail to distinguish between `public` and `private`, leading to security or performance issues.
- Ignore advanced directives like `s-maxage` or `stale-while-revalidate`.
- Don't realise that CDNs, browsers, proxies, and application caches all layer their own rules on top.

The result? Countless sites ship with fragile, inconsistent, or outright broken caching policies. They leave money on the table in infrastructure costs, frustrate users with

sluggish performance, and collapse under load that better-configured systems would sail through.

This guide exists to fix that. Over the next chapters, we'll unpack the ecosystem of HTTP caching in detail:

- How headers like `Cache-Control`, `Expires`, `ETag`, and `Age` actually work, alone and together.
- How browsers, CDNs, and app-level caches interpret and enforce them.
- The common pitfalls and misconceptions that can trip up even experienced developers.
- Practical recipes for static assets, HTML documents, APIs, and more.
- Modern browser behaviours, like BFCache, speculation rules, and signed exchanges.
- CDN realities, with a deep dive into Cloudflare's defaults, quirks, and advanced features.
- How to debug and verify caching in the real world.

By the end, you'll not only understand the nuanced interplay of HTTP caching headers – you'll know how to design and deploy a caching strategy that makes your sites faster, cheaper, and more reliable.

The business case for caching

Caching matters because it directly impacts four fundamental outcomes of how a site performs and scales:

Speed

Caching eliminates unnecessary network trips. A memory-cache hit in the browser is effectively instant, compared to the 100–300ms you'd otherwise wait just to complete

a handshake and see the first byte. Multiply that by dozens of assets and you get smoother page loads, lower Core Web Vitals, and happier users.

Resilience

When demand surges, cache hits multiply capacity. If 80% of traffic is absorbed by a CDN edge, your servers only need to handle the other 20%. That's the difference between sailing through Black Friday and collapsing under a viral traffic spike.

Cost

Every cache hit is one less expensive origin request. CDN bandwidth is cheap; uncached origin hits consume CPU, database queries, and outbound traffic that you pay for. A 5–10% improvement in cache hit ratio can translate directly into thousands of dollars saved at scale. And that's not even counting when requests are cached in users' browsers, and don't even hit the CDN!

SEO

Caching improves both speed and efficiency for search engines. Bots are less aggressive when they see effective caching headers, conserving crawl budget for fresher and deeper content. Faster pages also feed directly into Google's performance signals.

Real-world Scenarios

- A news site avoids a meltdown during a breaking story because 95% of requests are served from the CDN cache.
- An API under sustained load continues to respond consistently thanks to `stale-if-error` and validator-based revalidation.
- An e-commerce platform handles Black Friday traffic smoothly because static assets and category pages are long-lived at the edge.

Side note on the philosophy of caching

It's worth acknowledging that there's a quiet anti-culture around caching. Some developers see it as a hack – a band-aid slapped over slow systems, masking deeper flaws in design or architecture. In an ideal world, every request would be cheap, every response instant, and caching wouldn't even be needed. And there's merit in that vision: designing systems to be inherently fast avoids the complexity and fragility that caching introduces.

In practice, most of us don't live in that world. Real systems face unpredictable spikes, long geographic distances, and sudden swings in demand. Even the best-architected applications benefit from caching as an amplifier. The key is balance: caching should never excuse poor underlying performance, but it should always be part of how you scale and stay resilient when traffic surges.

Mental model: who caches what?

Before diving into the fine-grained details of headers and directives, it helps to understand the landscape of *who* is actually caching your content. Caching isn't a single thing that happens in one place—it's an ecosystem of layers, each with its own rules, scope, and quirks.

Browsers

Every browser maintains both a memory cache and a disk cache. The memory cache is extremely fast but short-lived – it only lasts while a page is open – and is designed to avoid redundant network fetches during a single session. It isn't governed by HTTP caching headers: even resources marked `no-store` may be reused from memory if they're requested again within the same page. The disk cache, by contrast, persists across tabs and sessions, can hold much larger resources, and does respect HTTP caching headers (though browsers may still apply their own heuristics when metadata is missing).

Proxies

Between the browser and the wider internet, requests often pass through proxies – especially in corporate environments or ISP-managed networks. These proxies can act as shared caches, storing responses to reduce bandwidth costs or to enforce organisational policies. Unlike CDNs, you usually don't configure them yourself, and their behaviour may be opaque.

For example, a corporate proxy might cache software downloads to avoid repeated gigabyte transfers across the same office connection. An ISP might cache popular news images to improve load times for customers. The problem is that these proxies don't always respect HTTP caching headers perfectly, and they may apply their own heuristics or overrides. That can lead to inconsistencies, like a user behind a proxy seeing a stale or stripped-down response long after it should have expired.

While less visible than browser or CDN caches, proxies are still an important part of the ecosystem. They remind us that caching isn't always under the site owner's direct control – and that intermediaries in the network can influence freshness, reuse, and even correctness.

> ### Side note on transparent ISP proxies
>
> In the early 2000s, many ISPs deployed "transparent" proxies that cached popular resources without users or site owners even knowing. They still crop up in some regions today. These proxies sit silently between the browser and the origin, caching opportunistically to save bandwidth. The downside is that they sometimes ignore cache headers entirely, serving outdated or inconsistent content. If you've ever seen a site behave differently at home vs on mobile data, a transparent proxy might have been the reason.

Shared caches

Between users and origin servers sit a host of shared caches – CDNs like Cloudflare or Akamai, ISP-level proxies, corporate gateways, or reverse proxies. These shared layers can dramatically reduce origin load, but they come with their own logic and sometimes override or reinterpret origin instructions.

Reverse proxies

Technologies like Varnish or NGINX can act as local accelerators in front of your application servers. They intercept and cache responses close to the origin, smoothing traffic spikes and offloading heavy lifting from your app or database.

Application and database caches

Inside your stack, systems like Redis or Memcached store fragments of rendered pages, precomputed query results, or sessions. They aren't governed by HTTP headers – you design the keys and TTLs yourself – but they are crucial parts of the caching ecosystem.

Cache keys and variants

Every cache needs a way to decide whether two requests are "the same thing" or not. That decision is made using a **cache key** – essentially, the unique identifier for a stored response.

By default, a cache key is based on the scheme, host, path, and query string of the requested resource. But in practice, browsers add more dimensions. Most implement **double-keyed caching**, where the top-level browsing context (the site you're on) is also part of the key. That's why your browser can't reuse a Google Font downloaded while visiting one site when another, unrelated site requests the same font file – each gets its own cache entry, even though the URL is identical.

Modern browsers are moving towards **triple-keyed caching**, which adds subframe context into the key as well. This means a resource requested inside an embedded iframe may have its own cache entry, separate from the same resource requested by the top-level page or by another iframe. This design improves privacy (by limiting cross-site tracking via shared cache entries), but it also reduces opportunities for cache reuse.

On top of that, HTTP adds another layer of complexity: the `Vary` header. This tells caches that *certain request headers* should also be part of the cache key.

Examples:

- `Vary: Accept-Encoding` → store one copy compressed with gzip, another with brotli.
- `Vary: Accept-Language` → store separate versions for `en-US` vs `de-DE`.
- `Vary: Cookie` → every unique cookie value creates a separate cache entry (often catastrophic).
- `Vary: *` → means "*you can't safely reuse this for anyone else*," which effectively kills cacheability.

This is powerful, and sometimes essential. If your server switches image formats based on `Accept` headers, or serves AVIF to browsers that support it, you *must* use `Vary: Accept` to avoid sending incompatible responses to clients that can't handle them. At the same time, `Vary` is easy to misuse. Carelessly adding `Vary: User-Agent`, `Vary: Cookie`, or `Vary: *` can explode your cache into thousands of near-duplicate entries. The key is to vary only on headers that genuinely change the response – nothing more.

That's where **normalisation** comes in. Smart CDNs and proxies can simplify cache keys, collapsing away differences that don't matter. For example:

- Ignoring analytics query parameters (e.g., `?utm_source=...`).
- Treating all iPhones as the same "mobile" variant, instead of keying on every device string.

The balance is to vary only on things that truly change the response. Anything else is wasted fragmentation and lower hit ratios.

> **Side note on `No-Vary-Search`**

> A new experimental header, `No-Vary-Search`, lets servers tell caches to ignore certain query parameters when deciding cache keys. For example, you could treat `?utm_source=` or `?fbclid=` as irrelevant and avoid fragmenting your cache into thousands of variants. At the moment, support is limited – Chrome only uses it with speculation rules – but if adopted more widely, it could offer a standards-based way to normalise cache keys without relying on CDN configuration.

Freshness vs validation

Knowing *who* is caching your content and *how* they decide whether two requests are the same only answers part of the question. The other part is **when a stored response can be reused**.

Every cache, whether it's a browser or a CDN, has to decide:

- Is this copy still **fresh** enough to serve as-is?
- Or has it gone **stale**, and do I need to check with the origin?

That's the core trade-off in caching: **freshness** (serve immediately, fast but risky if outdated) versus **validation** (double-check with the origin, slower but guaranteed correct).

All the headers we'll explore next – HTTP headers like `Cache-Control`, `Expires`, `ETag`, and `Last-Modified` help us to guide this decision-making process.

Core HTTP caching headers

Now that we know who caches content and how they make basic decisions, it's time to look at the raw materials: the headers that control caching. These are the levers you pull to influence every layer of the system – browsers, CDNs, proxies, and beyond.

At a high level, there are three categories:

- **Freshness controls:** tell caches how long a response can be served without revalidation.
- **Validators:** provide a way to check cheaply if something has changed.
- **Metadata:** describe how the response should be stored, keyed, or observed.

Let's break them down.

The `Date` header

Every response should carry a `Date` header. It's the server's timestamp for when the response was generated, and it's the baseline for all freshness and age calculations. If `Date` is missing or skewed, caches will make their own assumptions.

The `Cache-Control` (response) header

This is the most important header – the control panel for how content should be cached. It carries multiple directives, split into two broad groups:

Freshness directives:

- `max-age`: how long (in seconds) the response is fresh.
- `s-maxage`: like `max-age`, but applies only to shared caches (e.g. CDNs). Overrides `max-age` there.
- `immutable`: signals that the resource will never change (ideal for versioned static assets).
- `stale-while-revalidate`: allows serving a stale response while fetching a fresh one in the background.
- `stale-if-error`: allows serving stale content if the origin is down or errors.

Storage/use directives:

- `public`: response may be stored by any cache, including shared ones.

- `private`: response may be cached only by the browser, not shared caches.
- `no-cache`: store, but revalidate before serving.
- `no-store`: do not store at all.
- `must-revalidate`: once stale, the response *must* be revalidated before use.
- `proxy-revalidate`: same, but targeted at shared caches.

The `Cache-Control` (request) header

Browsers and clients can also send caching directives. These don't change the server's headers, but they influence how caches along the way behave.

- `no-cache`: forces revalidation (but allows use of stored entries).
- `no-store`: bypasses caching entirely.
- `only-if-cached`: instructs to return a cached response if available, otherwise error (useful offline).
- `max-age`, `min-fresh`, `max-stale`: fine-tune tolerance for staleness.

The `Expires` header

An older way of defining freshness, based on providing an absolute date/timestamp.

- Example: `Expires: Wed, 29 Aug 2025 12:00:00 GMT`.
- Ignored if `Cache-Control: max-age` is present.
- Vulnerable to clock skew between servers and clients.
- Still widely seen, often for backwards compatibility.

The `Pragma` header

The `Pragma` header dates back to HTTP 1.0 and was used to prevent caching before `Cache-Control` existed (on *requests*; asking intermediaries to revalidate content

before reuse). Modern browsers and CDNs now rely on `Cache-Control`, but some intermediaries and older systems still respect `Pragma`. In theory, it could take any arbitrary name/value pairs; in practice, only one ever mattered: `Pragma: no-cache`.

For maximum compatibility – especially when dealing with mixed or legacy infrastructure – it's harmless to include both.

The `Age` header

`Age` tells you how old the response is (in seconds) when delivered. It's supposed to be set by shared caches, but not every intermediary implements it consistently. Browsers never set it. Treat it as a helpful signal, not an absolute truth.

> ### Side note on `Age`
>
> You'll only ever see `Age` headers from *shared caches* like CDNs or proxies. Why? Because browsers don't expose their internal cache state to the network – they just serve responses directly to the user. Shared caches, on the other hand, need to communicate freshness downstream (to other proxies, or to browsers), so they add `Age`. That's why you'll often see `Age: 0` on a fresh CDN hit, but never on a pure browser cache hit.

Validator headers: `ETag` and `Last-Modified`

When freshness runs out, caches use validators to avoid re-downloading the whole resource.

- **`ETag`**: a unique identifier (opaque string) for a specific version of a resource.
  - Strong ETags (`"abc123"`) mean byte-for-byte identical.
  - Weak ETags (`W/"abc123"`) mean semantically the same, though bytes may differ (e.g. re-gzipped).
- **`Last-Modified`**: timestamp of when the resource last changed.
  - Less precise, but still useful.

    - Supports heuristic freshness when `max-age` / `Expires` are missing.
- **Conditional requests:**
    - `If-None-Match` (with ETag) → server replies `304 Not Modified` if unchanged.
    - `If-Modified-Since` (with Last-Modified) → same, but based on date.
    - Both save bandwidth and reduce load, because only headers are exchanged.

Side note on strong vs weak ETags

An ETag is an identifier for a specific version of a resource. A **strong ETag** (`"abc123"`) means *byte-for-byte identical* – if even a single bit changes (like whitespace), the ETag must change. A **weak ETag** (`W/"abc123"`) means *"semantically the same"* – the content may differ in trivial ways (e.g. compressed differently, reordered attributes) but is still valid to reuse.

Strong ETags give more precision, but can cause cache misses if your infrastructure (say, different servers behind a load balancer) generates slightly different outputs. Weak ETags are more forgiving, but less strict. Both work with conditional requests – the choice is about balancing precision vs practicality.

Side note on ETags vs `Cache-Control` headers

`Cache-Control` directives are processed *before* the ETag. If it determines that a resource is stale, the cache uses the ETag (or `Last-Modified`) to revalidate with the origin. Think of it this way:

While fresh: the cache serves the copy immediately, no validation.
When stale: the cache sends `If-None-Match: "etag-value"` .

If the origin replies `304 Not Modified` , the cache can keep using the stored copy without re-downloading the whole thing. Without `Cache-Control` , the ETag may be used for heuristic freshness or unconditional revalidation – but that usually means more frequent trips back to the origin. The two are designed to work together: `Cache-Control` sets the lifetime, ETags handle the check-ups.

The `Vary` header

The `Vary` header tells caches which request headers should be factored into the cache key. It's what allows a single URL to have multiple valid cached variants. For example, if a server responds with `Vary: Accept-Encoding`, the cache will store one copy compressed with `gzip` and another compressed with `brotli`. Each encoding is treated as a distinct object, and the right one is chosen depending on the next request.

This flexibility is powerful, but also easy to misuse. Setting `Vary: *` is effectively the same as saying "*this response can never be reused safely for anyone else*", which makes it uncacheable in shared caches. Similarly, `Vary: Cookie` is notorious for destroying hit rates, because every unique cookie value creates a separate cache entry.

The best approach is to keep `Vary` minimal and intentional. Only vary on headers that truly change the response in a meaningful way. Anything else just fragments your cache, lowers efficiency, and adds unnecessary complexity.

Observability helpers

Modern caches don't just make decisions silently – they often add their own debugging headers to help you understand what happened. The most important of these is `Cache-Status`, a new standard that reports whether a response was a `HIT` or a `MISS`, how long it sat in cache, and sometimes even why it was revalidated. Many CDNs and proxies also use the older `X-Cache` header for the same purpose, typically showing a simple `HIT` or `MISS` flag. Cloudflare goes a step further with its `cf-cache-status` header, which distinguishes between `HIT`, `MISS`, `EXPIRED`, `BYPASS` and `DYNAMIC` (and other values).

These headers are invaluable when tuning or debugging, because they reveal the cache's own decision-making rather than just echoing your origin's intent. A response

might look cacheable on paper, but if you see a steady stream of `MISS` or `DYNAMIC`, it probably means that the intermediary isn't following your headers the way you expect.

Freshness & age calculations

Once you understand who caches content and which headers control their behaviour, the next step is to see how those pieces come together in practice. Every cache – whether it's a browser, a CDN, or a reverse proxy – follows the same logic:

1. Work out how long the response should be considered **fresh**.
2. Work out how **old** the response currently is.
3. Compare the two, and decide whether to **serve**, **revalidate**, or **fetch** anew.

This is the hidden math that drives every "cache hit" or "cache miss" you'll ever see.

Freshness lifetime

The *freshness lifetime* tells a cache how long it can serve a response without re-checking with the origin. To work that out for a given request, caches look for the following HTTP response headers in a strict order of precedence:

1. **`Cache-Control: max-age`** (or **`s-maxage`**) → overrides everything else.
2. **`Expires`** → an absolute date, used only if `max-age` is absent.
3. **Heuristic freshness** → if neither of those directives is present, caches *guess*.

Example 1: `max-age`

```
Date: Tue, 29 Aug 2025 12:00:00 GMT
Cache-Control: max-age=300
```

Here, the server explicitly tells caches, *"This response is good for 300 seconds after the `Date`"*. That means the response can be considered fresh until **12:05:00 GMT**. After

that, it becomes stale unless revalidated.

Example 2: `Expires`

```
Date: Tue, 29 Aug 2025 12:00:00 GMT
Expires: Tue, 29 Aug 2025 12:10:00 GMT
```

There's no `max-age`, but `Expires` provides an absolute cutoff. Caches compare the `Date` (12:00:00) with the `Expires` time (12:10:00). That's a 10-minute freshness window: the response is fresh until **12:10:00**, then stale.

Example 3: Heuristic

```
Date: Tue, 29 Aug 2025 12:00:00 GMT
Last-Modified: Mon, 28 Aug 2025 12:00:00 GMT
```

With no `max-age` or `Expires`, caches fall back to heuristics. Browsers have varying approaches; Chrome uses *10% of the time since the last modification*. Here, the resource was last modified 24 hours ago, so the cache should be considered fresh for **2.4 hours** (until about 14:24:00 GMT), after which revalidation kicks in.

Current age

The current age is the cache's estimate of how old the response is right now. The spec gives a formula, but we can break it into steps:

- **Apparent age** = now – `Date` (if positive).
- **Corrected age** = max(Apparent age, `Age` header).
- **Resident time** = how long it's been sitting in the cache.
- **Current age** = Corrected age + Resident time.

Example 4: Simple case

```
Date: Tue, 29 Aug 2025 12:00:00 GMT
Cache-Control: max-age=60
```

The response was generated at 12:00:00 and reached the cache at 12:00:05, so it already appeared to be 5 seconds old when it arrived. With no `Age` header present, the cache then held onto it for another 15 seconds, making the total current age 20 seconds. Since the response had a `max-age` of 60 seconds, it was still considered fresh.

Example 5: With `Age` header

```
Date: Tue, 29 Aug 2025 12:00:00 GMT
Age: 30
Cache-Control: max-age=60
```

The origin sends a response stamped with `Date: 12:00:00` and also includes `Age: 30`, meaning some upstream cache already held it for 30 seconds. When a downstream cache receives it at 12:00:40, it looks 40 seconds old. The cache takes the higher of the two (40 vs 30) and then adds the 20 seconds it sits locally until 12:01:00. That makes the total current age 60 seconds – exactly matching the `max-age=60` limit. At that point, the response is no longer fresh and must be revalidated.

Decision tree

Once a cache knows both numbers:

- **If current age < freshness lifetime** → Serve immediately (fresh hit).
- **If current age ≥ freshness lifetime** →
  - If `stale-while-revalidate` → Serve stale now, revalidate it in the background.
  - If `stale-if-error` and origin is failing → Serve stale.
  - Else → Revalidate with origin (conditional GET/HEAD).

Example 6: `stale-while-revalidate`

```
Cache-Control: max-age=60, stale-while-revalidate=30
```

A response has `Cache-Control: max-age=60, stale-while-revalidate=30`. At 12:01:10, the cache's copy is 70 seconds old – 10 seconds beyond its freshness window. Normally, that would require a revalidation before serving, but `stale-while-revalidate` allows the cache to serve the stale copy instantly as long as it revalidates in the background. Because the copy is only 10 seconds into its 30-second stale allowance, the cache can safely serve it while updating in parallel.

Example 7: stale-if-error

```
Cache-Control: max-age=60, stale-if-error=600
```

Another response has `Cache-Control: max-age=60, stale-if-error=600`. At 12:02:00, the copy is 120 seconds old – well past its 60-second freshness lifetime. The cache tries to fetch a fresh version, but the origin returns a 500 error. Thanks to `stale-if-error`, the cache is allowed to fall back to its stale copy for up to 600 seconds while the origin remains unavailable, ensuring the user still gets a response.

Why this matters

Understanding the math explains a lot of "weird" behaviour:

- A resource expiring "too soon" may be down to a short `max-age` or a non-zero `Age` header.
- A response that looks stale but is served anyway may be covered by `stale-while-revalidate` or `stale-if-error`.
- A `304 Not Modified` doesn't mean caching failed – it means the cache correctly revalidated and saved bandwidth.

Caches aren't mysterious black boxes. They're just running these calculations thousands of times per second, across millions of resources. Once you know the math,

the behaviour becomes predictable – and controllable. But in practice, developers often trip over subtle defaults and misleading directive names. Let's tackle those misconceptions head-on.

Common misconceptions & gotchas

Even experienced developers misconfigure caching all the time. The directives are subtle, the defaults are quirky, and the interactions are easy to misunderstand. Here are some of the most common traps.

`no-cache` ≠ "don't cache"

The name is misleading. `no-cache` actually means *"store this, but revalidate before reusing it."* Browsers and CDNs will happily keep a copy, but they'll always check back with the origin before serving it. If you truly don't want anything stored, you need `no-store`.

`no-store` means nothing is kept

`no-store` is the nuclear option. It instructs every cache – browser, proxy, CDN – not to keep a copy at all. Every request goes straight to the origin. This is correct for highly sensitive data (e.g. banking), but overkill for most use cases. Many sites use it reflexively, throwing away huge performance gains.

`max-age=0` vs `must-revalidate`

They seem similar, but aren't the same. `max-age=0` means "*this response is immediately stale*". Without `must-revalidate`, caches are technically allowed to reuse it briefly under some conditions (e.g. if the origin is temporarily unavailable). Adding `must-revalidate` removes that leeway, forcing caches to always check with the origin once freshness has expired.

`s-maxage` vs `max-age`

`max-age` applies everywhere – browsers and shared caches alike. `s-maxage` only applies to shared caches like CDNs or proxies, and it **overrides** `max-age` there. This lets you set a short freshness window for browsers (say, `max-age=60`) but a longer one at the CDN (`s-maxage=600`). Many developers don't realise `s-maxage` even exists.

`immutable` misuse

`immutable` tells browsers *"this resource will never change, don't bother revalidating it".* That's fantastic for fingerprinted assets (like `app.9f2d1.js`) that are versioned by filename. But it's dangerous for HTML or any resource that might change under the same URL. Use it on the wrong thing, and you'll lock users into stale content for months.

Redirect and error caching

Caches can and do store redirects and even error responses. A `301` is cacheable by default (often permanently). Even a `404` or `500` may be cached briefly, depending on headers and CDN settings. Developers are often surprised when "temporary" outages linger because an error response was cached.

Clock skew and heuristic surprises

Caches compare `Date`, `Expires`, and `Age` headers to decide freshness. If clocks aren't perfectly in sync, or if no explicit headers are present, caches fall back to heuristics. That can lead to surprising expiry behaviour. Explicit freshness directives are always safer.

Cache fragmentation: devices & geography

Caching is simple when one URL maps to one response. It gets tricky when responses vary by device or region.

- **Device splits:** Sites often serve different HTML or JS for desktop vs mobile. If keyed on `User-Agent`, every browser/version combination becomes a separate cache entry; the result is that cache hit rates collapse. Safer options include normalising User-Agents at the CDN, or using Client Hints (`Sec-CH-UA`, `DPR`) with controlled `Vary` headers.
- **Geo splits:** Serving different content by region (e.g. India vs Germany) often uses `Accept-Language` or GeoIP rules. But every language combination (`en`, `en-US`, `en-GB`) creates a new cache key. Unless you normalise by region/ruleset, your cache fragments into thousands of variants.

The trade-off is clear: more personalisation usually means less caching efficiency. Once the traps are clear, we can move from theory to practice. Here are the caching "recipes" you'll use for different content types.

Patterns & recipes

Now that we've covered the mechanics and the common pitfalls, let's look at how to put caching into practice. These are the patterns you'll reach for again and again, adapted for different kinds of content.

Static assets (JS, CSS, fonts)

Goal: Serve instantly, never revalidate, safe to cache for a very long time.

Typical headers:

```
Cache-Control: public, max-age=31536000, immutable
```

Why:

- Fingerprinted filenames (`app.9f2d1.js`) guarantee uniqueness, so old versions can stay cached forever.

- Long `max-age` means they never expire in practice.
- `immutable` stops browsers from wasting time revalidating.

HTML documents

The right TTL depends on how often your HTML changes and how quickly changes must appear. Use one of these profiles, and pair long edge TTLs with **event-driven purge** on publish/update.

Profile A: High-change (news/homepages):

```
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=60, stale-i
ETag: "abc123"
```

Rationale: keep browsers very fresh (1m), let the CDN cushion load for 5m, serve briefly stale while revalidating for snappy UX, and survive origin wobbles.

Profile B – Low-change (blogs/docs):

```
Cache-Control: public, max-age=300, s-maxage=86400, stale-while-revalidate=300, sta
ETag: "abc123"
```

Rationale: browsers can reuse for a few minutes; CDN can hold for **a day** to slash origin traffic. On publish/edit, purge the page (and related pages) to make changes instantaneously.

Logged-in / personalised pages:

```
Cache-Control: private, no-cache
ETag: "abc123"
```

Rationale: allow browser storage but force revalidation every time; never share at the CDN.

> ### Side note on long HTML TTLs are safe with event-driven purge
>
> You can run **very long CDN** cache expiration times (hours, even days) for HTML as long as you **actively bust the cache** on important events: publish, update, unpublish. Use CDN features like **Cache Tags / surrogate keys** to purge collections (e.g., “post-123”, “author-jono”), and trigger purges from your CMS. This gives you the best of both worlds: instant updates when it matters, rock-solid performance the rest of the time.

If updates must appear within **seconds** with no manual purge → keep **short CDNs TTLs** (≤5m) + `stale-while-revalidate`.

If updates are **event-driven** (publish/edit) → use **long CDNs TTLs** (hours/days) + **automatic purge by tag**.

If content is **personalised** → **don’t share** (use `private, no-cache` + validators).

APIs

Goal: Balance freshness with performance and resilience.

Typical headers:

```
Cache-Control: public, s-maxage=30, stale-while-revalidate=30, stale-if-error=300
ETag: "def456"
```

Why:

- Shared caches (CDNs) can serve results for 30s, reducing load.
- `stale-while-revalidate` keeps latency low even as responses are refreshed.
- `stale-if-error` ensures reliability if the backend fails.
- Clients can revalidate cheaply with ETags.

Side note on why APIs use short `s-maxage` + `stale-while-revalidate`

APIs often serve data that changes frequently, but not *every single second*. A short `s-maxage` (e.g. 30s) lets shared caches like CDNs soak up most requests, while still ensuring data stays reasonably fresh.

Adding `stale-while-revalidate` smooths over the edges: even if the cache has to fetch a new copy, it can serve the slightly stale one instantly while revalidating in the background. That keeps latency low for users.

The combination gives you a sweet spot: **low origin load, fast responses, and data that's "fresh enough" for most real-world use cases.**

Authenticated dashboards & user-specific pages

Goal: Prevent shared caching, but allow browser reuse.

Typical headers:

```
Cache-Control: private, no-cache
ETag: "ghi789"
```

Why:

- `private` ensures only the end-user's browser caches the response.
- `no-cache` allows reuse, but forces validation first.
- ETags prevent full downloads on every request.

Side note on the omission of `max-age`

For user-specific content, you can't risk serving stale data. That's why the recipe uses `private, no-cache` but leaves out `max-age`.

- `no-cache` means the browser may keep a local copy, but must revalidate it with the origin before reusing it.
- If you added `max-age`, you'd be telling the browser it's safe to serve without checking – which could expose users to out-of-date account info or shopping carts.
- Pairing `no-cache` with an `ETag` gives you the best of both worlds: safety (always validated) and efficiency (cheap 304 Not Modified responses instead of re-downloading everything).

Side note on security

When handling or presenting sensitive data, you *may* wish to use `private, no-store` instead, in order to prevent the browser from storing a locally available cached version. This reduces the likelihood of leaks on devices used by multiple users, for example.

Images & media

Goal: Cache efficiently across devices, while serving the right variant.

Typical headers:

```
Cache-Control: public, max-age=86400
Vary: Accept-Encoding, DPR, Width
```

Why:

- A one-day freshness window balances speed with flexibility – images can change, but not as often as HTML.
- `Vary` allows different versions to be cached for different devices or display densities.
- CDNs can normalise query parameters (e.g. ignore `utm_*`) and collapse variants intelligently to avoid fragmentation.

Side note on client hints

Modern browsers send **Client Hints** like `DPR` (device pixel ratio) and `Width` (intended display width) when requesting images. If your server or CDN supports responsive images, it can generate and return different variants—e.g. a high-res version for a retina phone, a smaller one for a low-res laptop.

By including `Vary: DPR, Width`, you're telling caches: *"Store separate copies depending on these hints."* That ensures the right variant is reused for future requests with the same device characteristics.

The catch? Every new `DPR` or `Width` value creates a new cache key. If you don't normalise (e.g. bucket widths into sensible breakpoints), your cache can fragment into hundreds of variants. CDNs often provide built-in rules to manage this.

Beyond headers: browser behaviours

HTTP headers set the rules, but browsers have their own layers of optimisation that can look like "caching" – or interfere with it. These don't follow the same rules as `Cache-Control` or `ETag`, and they often confuse developers when debugging.

Back/forward cache (BFCache)

- **What it is:** A full-page snapshot (DOM, JS state, scroll position) kept in memory when a user navigates away.
- **Why it matters:** Going "back" or "forward" feels instant because the browser restores the page without even hitting HTTP caches.
- **Gotchas:** Many pages aren't BFCache-eligible. The most common blockers are unload handlers, long-lived connections, or the use of certain browser APIs. Another subtle but important one is Cache-Control: no-store on the document itself – this tells the browser not to keep any copy around, which extends to BFCache. Chrome

has recently carved out a small set of exceptions where no-store pages can still enter BFCache in safe cases, but for the most part, if you want BFCache eligibility, you should avoid `no-store` on documents.

> **Side note on BFCache vs HTTP Cache**
>
> BFCache is like pausing a tab and resuming it – the entire page state is frozen and restored. HTTP caching only stores network resources. A page might fail BFCache but still be quite fast thanks to HTTP cache hits (or vice versa).

Hard refresh vs soft reload

- **Soft reload** (e.g. pressing the reload button): Browser will use cached responses if they're still fresh. If stale, it revalidates.
- **Hard refresh** (e.g. opening DevTools and right-clicking the reload button to do a fuller reload, or ticking the "disable cache" button): Browser bypasses the cache, re-fetching all resources from the origin.
- **Gotcha:** Users may think "refresh" always fetches new content – but unless it's a hard refresh, caches still apply.

Speculation rules & link relations

Browsers provide developers with tools that let them (pre)load resources, before the user requests them. These don't change how caching *works*, but they *can* change what ends up *in* the cache ahead of time.

- **Prefetch:** The browser may fetch resources speculatively and place them in cache, but only for a short window. If they're not used quickly, they'll be evicted.
- **Preload:** Resources are fetched early and inserted into cache so they're ready by the time the parser needs them.
- **Prerender:** The entire page and its subresources are loaded and cached in advance. When a user navigates, it all comes straight from cache rather than the network.

- **Speculation rules API:** Eviction, freshness, and validation usually follow the normal caching rules – but prerendering makes some exceptions. For example, Chrome may prerender a page even if it's marked with Cache-Control: no-store or no-cache. In those cases, the prerendered copy lives in a temporary store that isn't part of the standard HTTP cache and is discarded once the prerender session ends (though this behaviour may vary by browser).

The key takeaway: **speculation rules are about cache timing, but not cache policy.** They front-load work so the cache is already warm, but freshness and expiry are still governed by your headers.

Signed exchanges (SXG)

Signed exchanges don't change cache mechanics either, but they do change **who can serve cached content** while keeping origin authenticity intact.

- An SXG is a package of a response, plus a cryptographic signature from the origin.
- Intermediaries (like Google Search) can store and serve that package from their *own* caches.
- When the browser receives it, it can trust the content as if it came from *your* domain, while still applying your headers for freshness and validation.

The catch: SXGs have their own **signature expiry** in addition to your normal caching headers. Even if your `Cache-Control` allows reuse, the SXG may be discarded once its signature is out of date.

SXGs also support **varying by cookie**, which means they can package and serve different signed variants depending on cookie values. This enables personalised experiences to be cached and distributed via SXG, but it fragments the cache heavily – every cookie combination creates a new variant.

Key takeaway: SXG adds another clock (signature lifetime) and, if you use cookie variation, another source of cache fragmentation. Your headers still govern freshness,

but these extra layers can shorten reuse windows and multiply cache entries.

CDNs in practice: Cloudflare

So far, we've looked at how browsers handle caching and the directives that control freshness and validation. But for most modern websites, the **first and most important cache** your traffic will hit isn't the browser—it's the CDN.

Cloudflare is one of the most widely used CDNs, fronting millions of sites. It's a great example of how shared caches don't just passively obey your headers. They add defaults, overrides, and proprietary features that can completely change how caching works in practice. Understanding these quirks is essential if you want your origin headers and your CDN behaviour to align.

Defaults and HTML Caching

By default, Cloudflare doesn't cache HTML at all. Static assets like CSS, JavaScript, and images are stored happily at the edge, but documents are always passed through to the origin unless you explicitly enable "Cache Everything." That default catches many site owners out: they assume Cloudflare is shielding their servers, when in reality their most expensive requests – the HTML pages themselves – are still hitting the backend every time.

The temptation, then, is to flip the switch and enable "Cache Everything." But this blunt tool applies indiscriminately, even to pages that vary by cookie or authentication state. In that scenario, Cloudflare can end up serving cached private dashboards or logged-in user data to the wrong people.

The safer pattern is more nuanced: bypass the cache when a session cookie is present, but cache aggressively when the user is anonymous. This approach ensures that public pages reap the benefits of edge caching, while private content is always fetched fresh from the origin.

> ### Side note on Cloudflare's APO addon
>
> Cloudflare's Automatic Platform Optimization (APO) addon integrates with WordPress websites, and rewrites caching behaviour so HTML can be cached safely while respecting logged-in cookies. It's a good example of CDNs layering platform-specific heuristics on top of standard HTTP logic.

Edge vs browser lifetimes

Your origin headers – things like `Cache-Control` and `Expires` – define how long a browser should hold onto a resource. But CDNs like Cloudflare add another layer of control with their own settings, such as "Edge Cache TTL" and `s-maxage`. These apply only to what Cloudflare stores at its edge servers, and they can override whatever the origin says without changing how the browser behaves.

That separation is both powerful and confusing. From the browser's perspective, you might see `max-age=60` and assume the content is cached for just a minute. Meanwhile, Cloudflare could continue serving the same cached copy for ten minutes, because its edge cache TTL is set to 600 seconds. The result is a split reality: browsers refresh often, but Cloudflare still shields the origin from repeated requests.

Cache keys and fragmentation

Cloudflare uses the full URL as its cache key. That means every distinct query parameter – whether it's a tracking token like `?utm_source=…` or something trivial like `?v=123` – creates a separate cache entry. Left unchecked, this behaviour quickly fragments your cache into hundreds of near-identical variants, each one consuming space while reducing the hit rate.

It's important to note that canonical URLs don't help here. Cloudflare doesn't care what your HTML declares as the "true" version of a page; it caches by the literal request URL it receives. To avoid fragmentation, you need to explicitly normalise or

ignore unnecessary parameters in Cloudflare's configuration, ensuring that trivial differences don't splinter your cache.

> **Site note on normalising cache keys**
>
> Cloudflare lets you define which query parameters to ignore, or how to collapse variants. Stripping out analytics paramaters, for example, can dramatically improve cache hit ratios.

Device and geography splits

Cloudflare also allows you to customise cache keys by including request headers, such as `User-Agent` or geo-based values. In theory, this enables fine-grained caching—one version of a page for mobile devices, another for desktop, or distinct versions for visitors in different countries.

But in practice, unless you normalise these inputs aggressively, it can explode into massive fragmentation. Caching by raw `User-Agent` means every browser and version string generates its own entry, instead of collapsing them into a simple "mobile vs desktop" split. The same problem arises with geographic rules: caching by full `Accept-Language` headers, for example, can create thousands of variants when only a handful of languages are truly necessary.

Done carefully, device and geography splits let you serve tailored content from cache. Done carelessly, they destroy your hit rate and multiply origin load.

Cache tags

Cloudflare also supports tagging cached objects with labels – for example, tagging every page of a blog post with `blog-post-123`. These tags allow you to purge or revalidate whole groups of resources at once, rather than expiring them one by one.

For CMS-driven sites, this is a powerful tool: when an article is updated, the site can trigger a purge for its tag and instantly invalidate every related URL. But over-tagging

– attaching too many labels to too many resources – is common, and can undermine efficiency and make purge operations slower or less predictable.

Other caching layers in the stack

So far, we've focused on browser caches, HTTP directives, and CDNs like Cloudflare. But many sites add even more layers between the user and the origin. Reverse proxies, application caches, and database caches all play a role in what a "cached" response actually means.

These layers don't always speak HTTP – Redis doesn't care about `Cache-Control`, and Varnish can happily override your origin headers. But they still shape the user experience, infrastructure load, and the headaches of cache invalidation. To understand caching in the real world, you need to see how these pieces stack and interact.

Application & database caches

Inside the application tier, technologies like **Redis** and **Memcached** are often used to keep session data, fragments of rendered pages, or precomputed query results. An ecommerce site, for example, might cache its "*Top 10 Products*" list in Redis for sixty seconds, saving hundreds of database queries every time a page loads. This is fantastically efficient – until it isn't.

One common failure mode is when the database updates, but the Redis key isn't cleared at the right moment. In that case, the HTTP layer happily serves "fresh" pages that are already out of date, because they're pulling from stale Redis data underneath.

The inverse problem happens just as often. Imagine the app has correctly refreshed Redis with a new product price, but the CDN or reverse proxy still has an HTML page cached with the old price. The origin told that the outer cache that the page was valid for five minutes, so until the TTL runs out (or someone manually purges it), users continue seeing stale HTML – even though Redis already has the update.

In other words: sometimes HTTP looks fresh while Redis is stale, and sometimes Redis is fresh while HTTP caches are stale. Both failure modes stem from the same root issue – multiple caching layers, each with its own logic, falling out of sync.

Reverse proxy caches

One layer closer to the edge, reverse proxies like **Varnish** or **NGINX** often sit in front of the application servers, caching whole responses. In principle, they respect HTTP headers, but in practice, they're usually configured to enforce their own rules.
A Varnish configuration might, for example, force a five-minute lifetime on all HTML pages, regardless of what the origin headers say. That's excellent for resilience during a traffic spike, but dangerous if the content is time-sensitive. Developers frequently run into this mismatch: they open DevTools, inspect the origin's headers, and assume they know what's happening – not realising that Varnish is rewriting the rules one hop earlier.

Service Workers

Service Workers add another cache layer inside the browser, sitting between the network and the page. Unlike the built-in HTTP cache, which just follows headers, the Service Worker Cache API is programmable. That means developers can intercept requests and decide – in JavaScript – whether to serve from cache, fetch from the network, or do something else entirely.

This is powerful: a Service Worker can precache assets during install, create custom caching strategies (stale-while-revalidate, network-first, cache-first), or even rewrite responses before handing them back to the page. It's the foundation of Progressive Web Apps (PWAs) and offline support.

But it comes with pitfalls. Because Service Workers can ignore origin headers and invent their own logic, they can drift out of sync with the HTTP caching layer. For example, you might set `Cache-Control: max-age=60` on an API, but a Service Worker coded to "cache forever" will happily serve stale results long after they should have

expired. Debugging gets trickier too: responses can look cacheable in DevTools but actually be served from a Service Worker's script.

The key takeaway: Service Workers don't replace HTTP caching – they stack on top of it. They give developers fine-grained control, but they also add another layer where things can go wrong if caching strategies conflict.

Layer interactions

The real complexity comes when all these layers interact. A single request might pass through the browser cache, then Cloudflare, then Varnish, and finally Redis. Each layer has its own rules about freshness and invalidation, and they don't always line up neatly. You might purge the CDN and think you've fixed an issue, but the reverse proxy continues to serve its stale copy. Or you might flush Redis and repopulate the data, only to discover the CDN is still serving the "old" version it cached earlier. These kinds of mismatches are the root cause of many mysterious "cache bugs" that show up in production.

Debugging & verification

With so many caching layers in play – browsers, CDNs, reverse proxies, application stores – the hardest part of working with caching is often figuring out *which cache* served a response and *why*. Debugging caching isn't about staring at a single header; it's about tracing requests through the stack and verifying how each layer is behaving.

Inspecting headers

The first step is to look closely at the headers. Standard fields like `Cache-Control`, `Age`, `ETag`, `Last-Modified`, and `Expires` tell you what the origin intended. But they don't tell you what the caches actually did. For that, you need the debugging signals added along the way:

- `Age` shows how long a response has been sitting in a shared cache. If it's `0`, the response likely came from origin. If it's `300`, you know a cache has been serving the same object for five minutes.
- `X-Cache` (used by many proxies) or `cf-cache-status` (Cloudflare) show whether a cache hit or miss occurred.
- `Cache-Status` is the emerging standard, adopted by CDNs like Fastly, which reports not just HIT/MISS but also *why* a decision was made.

Together, these headers form the breadcrumb trail that tells you where the response has been.

Using browser DevTools

The Network panel in Chrome or Firefox's DevTools is essential for seeing cache behaviour from the user's side. It shows whether a resource came from disk cache, memory cache, or over the network.

- **Memory cache** hits are near-instant but short-lived, surviving only within the current tab/session.
- **Disk cache** hits persist across sessions but may be evicted.
- **304 Not Modified** responses reveal that the browser revalidated the cached copy with the origin.

It's also worth testing with different reload types. A normal reload (`Ctrl+R`) may use cached entries, while a hard reload (`Ctrl+Shift+R`) bypasses them entirely. Knowing which type of reload you're performing avoids false assumptions about what the cache is doing.

CDN logs and headers

If you're using a CDN, its logs and headers are often the most reliable source of truth. Cloudflare's `cf-cache-status`, Akamai's `X-Cache`, and Fastly's `Cache-Status` headers

all reveal edge decisions. Most providers also expose logs or dashboards where you can see hit/miss ratios and TTL behaviour at scale.

For example, if you see `cf-cache-status: MISS` or `BYPASS` on every request, it usually means Cloudflare isn't storing your HTML at all – either because it's following defaults (no HTML caching), or because a cookie is bypassing cache. Debugging at the edge often comes down to correlating what your origin sent, what the CDN says it did, and what the browser eventually received.

Reverse proxies and custom headers

Reverse proxies like Varnish or NGINX can be more opaque. Many deployments add custom headers like `X-Cache: HIT` or `X-Cache: MISS` to reveal proxy behaviour. If those aren't available, logs are your fallback: Varnish's `varnishlog` and NGINX's access logs can both show whether a request was served from cache or passed through.

The tricky part is remembering that reverse proxies may override headers silently. If you see `Cache-Control: no-cache` from origin but a five-minute TTL in Varnish, the headers in DevTools won't tell you the full story. You need the proxy's own debugging signals to verify.

Following the request path

When in doubt, step through the request chain:

1. **Browser** → Check DevTools: was it memory, disk, or network?
2. **CDN** → Inspect `cf-cache-status`, `Cache-Status`, or `X-Cache`.
3. **Proxy** → Look for custom headers or logs to confirm whether the request hit local cache.
4. **Application** → See if Redis/Memcached served the data.
5. **Database** → If all else fails, confirm the query ran.

Walking layer by layer helps isolate where the stale copy lives. It's rarely the case that "the cache is broken." More often, *one cache* is misaligned while the others are behaving perfectly.

Common debugging mistakes

There are a few traps developers fall into repeatedly:

- **Only looking at browser headers:** These tell you what the origin *intended*, not what the CDN actually did.
- **Assuming `304 Not Modified` means no caching:** In fact, it means the cache *did* store the response and successfully revalidated it.
- **Forgetting about cookies:** A stray cookie can make a CDN bypass cache entirely.
- **Testing with hard reloads:** A hard reload bypasses the cache, so it doesn't reflect normal user experience. The same is true if you enable the "Disable cache" tickbox in DevTools – that setting forces every request to skip caching entirely while DevTools is open. Both are useful for troubleshooting, but they give you an artificial view of performance that real users will never see.
- **Ignoring multi-layer conflicts:** Purging the CDN but forgetting to clear Varnish, or clearing Redis but leaving a stale copy at the edge.

Good debugging is less about clever tricks and more about being systematic: check each layer, verify its decision, and compare against what you expect from the headers.

Caching in the AI-mediated web

Up to now, we've treated caching as a conversation between websites, browsers, and CDNs. But increasingly, the consumers of your site aren't human users at all – they're search engine crawlers, LLM training pipelines, and agentic assistants. These systems rely heavily on caching, and your headers can shape not just performance, but how your brand and content are represented in machine-mediated contexts.

Crawl & scrape efficiency

Search engines and scrapers rely on HTTP caching to avoid re-downloading the entire web every day. Misconfigured caching can make crawlers hammer your origin unnecessarily, or worse, cause them to give up on deeper pages if revalidation is too costly. Well-tuned headers keep crawl efficient and ensure that fresh updates are discovered quickly.

Training data freshness

LLMs and recommendation systems ingest web content at scale. If your resources are always marked `no-store` or `no-cache`, they may get re-fetched inconsistently, leading to patchy or outdated snapshots of your site in training corpora. Conversely, stable cache policies help ensure that what makes it into these models is consistent and representative.

Agentic consumption

In an AI-mediated web, agents may act on behalf of users – shopping bots, research assistants, travel planners. For these agents, speed and reliability are first-class signals. A site with poor caching may look slower or less consistent than its competitors, biasing agents away from recommending it. In this sense, caching isn't just about performance for humans – it's about competitiveness in machine-driven decision-making.

Fragmentation risks

If caches serve inconsistent or fragmented variants – split by query strings, cookies, or geography – that noise propagates into machine understanding. A crawler or model might see dozens of subtly different versions of the same page. The result isn't just poor cache efficiency; it's a fractured representation of your brand in training data and agent outputs.

Wrapping up: caching as strategy

Caching is often treated as a technical detail, an afterthought, or a hack that papers over performance problems. But the truth is more profound: caching is infrastructure. It's the nervous system that keeps the web responsive under load, that shields brittle origins, and that shapes how both humans and machines experience your brand.

When it's configured badly, caching makes sites slower, more fragile, and more expensive. It fragments user experience, confuses crawlers, and poisons the well for AI systems that are already struggling to understand the web. When it's configured well, it's invisible—things just feel fast, resilient, and trustworthy.

That's why caching can't just be left to chance or to defaults. It needs to be a deliberate strategy, as fundamental to digital performance as security or accessibility. A strategy that spans layers—browser, CDN, proxy, application, database. A strategy that understands not just how to shave milliseconds for a single user, but how to present a coherent, consistent version of your site to millions of users, crawlers, and agents simultaneously.

The web isn't getting simpler. It's getting faster, more distributed, more automated, and more machine-mediated. In that world, caching isn't a relic of the old performance playbook. It's the foundation of how your site will scale, how it will be perceived, and how it will compete.

Caching is not an optimisation. It's a strategy.

Join the discussion

🙂

Name* Name*

Email* Email*

Post Comment

15 Comments

Inline Feedbacks

View all comments

Aleksei

Oct 14, 2025 11:05 pm

There is also “raced” requests, RCWN (Race Cache With Network) https://slides.com/valentin-gosu/race-cache-with-network-2017

A firefox optimization for cases when disk is slower than network (yes it is real)

Reply

Jono Alderson

Reply to Aleksei Oct 15, 2025 10:01 am

That’s fun, thanks for sharing!

Reply

Patrick Kerschbaum

Oct 13, 2025 9:56 am

Feature request:

At hokify.at we link to this guide in some places now, maybe you could consider copy-deep-links buttons/links for headings for such long articles (we want to link to the “recipes” of this article specifically).

Or Table-Of-Contents

Sure I can extract the ids myselves to construct urls like https://www.jonoalderson.com/performance/http-caching#h-static-assets-js-css-fonts, but still 🙂

Reply

Jono Alderson

Reply to Patrick Kerschbaum Oct 13, 2025 5:07 pm

Good idea, ~~I'll add it to my to-do list~~ done!

Reply

Patrick Kerschbaum

Reply to Jono Alderson Oct 26, 2025 3:40 pm

👍

Reply

R8S6N

Oct 11, 2025 9:07 am

Just add 'use cache', and problem solved 🙂

Reply

Patrick Kerschbaum

Reply to R8S6N Oct 13, 2025 9:58 am

Are you referring to Next.js 'use cache' directive? https://nextjs.org/docs/app/api-reference/directives/use-cache if so, this is very framework specific, while this article will teach you how HTTP caching on the web works.

Reply

Kieran

Oct 6, 2025 4:59 am

Might also pay to mention the (almost) long forgotten HTTP 1.0 HTTP response header "Pragma", which quite a few articles still reference. For maximum compatibility with different CDNs, proxies, etc, this is what I use for data containing dynamic data:

Cache-Control: private, no-store
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Pragma: no-cache

And this is what I use for CSS, Images, etc that are fingerprinted:

Cache-Control: public, max-age=31536000, immutable

Expires: Mon, 06 Oct 2125 00:00:00 GMT

Reply

Jono Alderson

Reply to Kieran Oct 6, 2025 9:15 pm

Ooh, pragma takes me back! Thanks, I've added a note!

Reply

Kieran

Oct 4, 2025 4:07 am

I would suggest a change (or a prominent info bubble) to "Authenticated dashboards & user-specific pages" section. You suggest "private, no-cache", which as you mentioned earlier in the article, allows the browser to store a cached copy but revalidate before using it. The issue is that the cache is then still on disk available to be accessed even if the user logs out (or their session times out). This is really bad for any site that handles private information, like PII, Finance, Medical, etc. In those cases, you must use "private, no-store". The risk of an information leak far outweighs the slight performance gain from "no-cache" (since in most cases, origins need to fully process the request before the final output can be revalidated).

Reply

Jono Alderson

Reply to Kieran Oct 4, 2025 2:41 pm

Hi! Good shout, added a note!

Reply

Patrick Kerschbaum

Oct 2, 2025 1:36 pm

This is a fantastic explanation of HTTP caching, thx so much for it!

One question: In section “Using browser DevTools” it says “304 Not Modified responses reveal that the browser revalidated with the origin before serving the cached copy.”
But in case of stale-while-revalidate being present in the cache-controlheader, the browser would serve the cached (and stale) copy immediately and revalidate in the background, right?

So maybe better is: “304 Not Modified responses reveal that the browser revalidated the cached copy with the origin.”

Reply

Jono Alderson

Reply to Patrick Kerschbaum Oct 2, 2025 2:52 pm

Ah, excellent nuance. Yes, thanks! Updated.

Reply

Vitor

Oct 2, 2025 12:13 pm

Such a great in depth article about cache. Thanks for sharing!

Reply

Jorge Jaroslavsky

Aug 29, 2025 5:27 pm

This is the most comprehensive explanation of caching headers I’ve seen, especially regarding their impact on SEO. Essential reading for anyone looking to boost site speed and search rankings! Many thanks for sharing.

Reply

I'm speaking at...

Web Directions Enqueue

2025-11-28
Sydney, Australia
webdirections.org/enqueue/

Experimentation Elite

2025-12-09
London, UK
experimentationelite.com

SMX Munich, 2026

2026-03-10 / 2026-03-11
Munich, Germany
smxmuenchen.de

Recent blog posts...

Optimising for the surfaceless web

30th October, 2025

The Hotmail effect

25th October, 2025

📣 Hire me!

I'm an award-winning consultant, and I'm available for hire.

Ready to take your website's **technical SEO**, **performance**, or **structured data strategy** to the next level?

Hire me as a consultant

Get in touch

Email: me@jonoalderson.com

Insert