Caching challenges and strategies

Matt Brinkley, Jas Chhabra

The ecstasy and the agony of caches

Over years of building services at Amazon we’ve experienced various versions of the following scenario: We build a new service, and this service needs to make some network calls to fulfill its requests. Perhaps these calls are to a relational database, or an AWS service like Amazon DynamoDB, or to another internal service. In simple tests or at low request rates the service works great, but we notice a problem on the horizon. The problem might be that calls to this other service are slow or that the database is expensive to scale out as call volume increases. We also notice that many requests are using the same downstream resource or the same query results, so we think that caching this data could be the answer to our problems. We add a cache and our service appears much improved. We observe that request latency is down, costs are reduced, and small downstream availability drops are smoothed over. After a while, no one can remember life before the cache. Dependencies reduce their fleet sizes accordingly, and the database is scaled down. Just when everything appears to be going well, the service could be poised for disaster. There could be changes in traffic patterns, failure of the cache fleet, or other unexpected circumstances that could lead to a cold or otherwise unavailable cache. This in turn could cause a surge of traffic to downstream services that can lead to outages both in our dependencies and in our service.

We’ve just described a service that has become addicted to its cache. The cache has been inadvertently elevated from a helpful addition to the service to a necessary and critical part of its ability to operate. At the heart of this issue is the modal behavior introduced by the cache, with differing behavior depending on whether a given object is cached. An unanticipated shift in the distribution of this modal behavior can potentially lead to disaster.

We have experienced both the benefits and challenges of caching in the course of building and operating services at Amazon. The remainder of this article describes our lessons learned, best practices, and considerations for using caches.

When we use caching

Several factors lead us to consider adding a cache to our system. Many times this begins with an observation about a dependency's latency or efficiency at a given request rate. For example, this could be when we determine that a dependency might start throttling or otherwise be unable to keep up with the anticipated load. We’ve found it helpful to consider caching when we encounter uneven request patterns that lead to hot-key/hot-partition throttling. Data from this dependency is a good candidate for caching if such a cache would have a good cache hit ratio across requests. That is, results of calls to the dependency can be used across multiple requests or operations. If each request typically requires a unique query to the dependent service with unique-per-request results, then a cache would have a negligible hit rate and the cache does no good. A second consideration is how tolerant a team’s service and its clients are to eventual consistency. Cached data necessarily grows inconsistent with the source over time, so caching can only be successful if both the service and its clients compensate accordingly. The rate of change of the source data, as well as the cache policy for refreshing data, will determine how inconsistent the data tends to be. These two are related to each other. For example, relatively static or slow-changing data can be cached for longer periods of time.

Local caches

Service caches can be implemented either in memory or external to the service. On-box caches, commonly implemented in process memory, are relatively quick and easy to implement and can provide significant improvements with minimal work. On-box caches are often the first approach implemented and evaluated once the need for caching is identified. In contrast to external caches, they come with no additional operational overhead, so they are fairly low-risk to integrate into an existing service. We often implement an on-box cache as an in-memory hash table that is managed through application logic (for example, by explicitly placing results into the cache after the service calls are completed) or embedded in the service client (for example, by using a caching HTTP client).

Despite the benefits and seductive simplicity of in-memory caches, they do come with several downsides. One is that the cached data will be inconsistent from server to server across its fleet, manifesting a cache coherence problem. If a client makes repeated calls to the service they might get newer data used in the first call and older data in the second call, depending on which server happens to handle the request.

Another shortcoming is that the downstream load is now proportional to the service's fleet size, so as the number of servers grows it still may be possible to overwhelm dependent services. We’ve found that an effective way to monitor this is to emit metrics on cache hits/misses and the number of requests made to downstream services.

In-memory caches are also susceptible to “cold start” issues. These issues occur where a new server launches with a completely empty cache, which could cause a burst of requests to the dependent service as it fills its cache. This can be a significant issue during deployments or in other circumstances in which the cache is flushed fleet-wide. Cache coherence and empty cache issues can often be addressed by using request coalescing, which is described in detail later in this article.

External caches

External caches can address many of the issues we’ve just discussed. An external cache stores cached data in a separate fleet, for example using Memcached or Redis. Cache coherence issues are reduced because the external cache holds the value used by all servers in the fleet. (Note that these issues aren’t totally eliminated because there might be failure cases when updating the cache.) Overall load on downstream services is reduced compared to in-memory caches and isn’t proportional to fleet size. Cold start issues during events like deployments are not present since the external cache remains populated throughout the deployment. Finally, external caches provide more available storage space than in-memory caches, reducing occurrences of cache eviction due to space constraints.

However, external caches come with their own set of shortcomings to consider. The first is an increased overall system complexity and operational load, since there is an additional fleet to monitor, manage, and scale. The availability characteristics of the cache fleet will be different from the dependent service it is acting as a cache for. The cache fleet can often be less available, for example, if it doesn’t have support for zero-downtime upgrades and if it requires maintenance windows.

To prevent having service availability degraded due to the external cache, we’ve found that we must add service code to deal with cache fleet unavailability, cache node failure, or cache put/get failures. One option is to fall back to calling the dependent service, but we’ve learned that we need to be careful when taking this approach. During an extended cache outage, this will cause an atypical spike in traffic to the downstream service, leading to throttling or brownout of that dependent service and ultimately reducing availability. We prefer to either use the external cache in conjunction with an in-memory cache that we can fall back to if the external cache becomes unavailable, or to use load shedding and cap the maximum rate of requests sent to the downstream service. We test service behavior with caching disabled to validate that the safeguards we've put in place to prevent the browning out of dependencies are actually working as expected.

A second consideration is the scaling and elasticity of the cache fleet. As the cache fleet begins to reach its request rate or memory limits, nodes will need to be added. We determine which metrics are leading indicators of these limits so we can set up monitors and alarms accordingly. For example, on a service I recently worked on, our team found that CPU utilization got very high as the Redis request rate reached its limit. We used load testing with realistic traffic patterns to determine the limit and find the right alarm threshold.

As we add capacity to the cache fleet, we take care to do so in a way that doesn't cause an outage or a massive loss of cache data. Different caching technologies have unique considerations. For example, some cache servers don't support adding nodes to a cluster without downtime, and not all cache client libraries provide consistent hashing, which is necessary to add nodes to the cache fleet and redistribute cached data. Due to the variability in client implementations of consistent hashing and the discovery of nodes in the cache fleet, we thoroughly test adding and removing cache servers before going to production.

With an external cache, we take extra care to ensure robustness as the storage format is changed. Cached data is treated as if it were in a persistent store. We ensure that updated software can always read data that a previous version of the software wrote, and that older versions can gracefully handle seeing new formats/fields (for example, during deployments when the fleet has a mix of old and new code). Preventing uncaught exceptions when unexpected formats are encountered is necessary to prevent poison pills. However, this isn't sufficient to prevent all format-related problems. Detecting a version format mismatch and discarding the cached data can lead to mass refreshes of caches, which can lead to dependent service throttling or brownouts. Serialization format issues are covered in greater depth in the Ensuring rollback safety during deployments article.

A final consideration for external caches is that they are updated by individual nodes in the service fleet. Caches typically don't have features like conditional puts and transactions, so we take care to ensure that the cache updating code is correct and can never leave the cache in an invalid or inconsistent state.

Inline vs. side caches

Another decision we need to make when we evaluate different caching approaches is the choice between inline and side caches. Inline caches, or read-through/write-through caches, embed cache management into the main data access API, making cache management an implementation detail of that API. Examples include application-specific implementations like Amazon DynamoDB Accelerator (DAX) and standards-based implementations like HTTP caching (either with a local caching client or an external cache server like Nginx or Varnish). Side caches, in contrast, are generic object stores such as the ones provided by Amazon ElastiCache (Memcached and Redis) or libraries like Ehcache and Google Guava for in-memory caches. With side caches, the application code directly manipulates the cache before and after calls to the data source, checking for cached objects before making the downstream calls, and putting objects in the cache after those calls are completed.

The primary benefit of an inline cache is a uniform API model for clients. Caching can be added, removed, or tweaked without any changes to client logic. An inline cache also pulls cache management logic out of application code, thus eliminating a source of potential bugs. HTTP caches are especially attractive because there are numerous off-the-shelf options available, such as in-memory libraries, standalone HTTP proxies like the ones mentioned previously, and managed services like content delivery networks (CDNs).

However, the transparency of inline caches can also be an availability downside. External caches are now part of the availability equation for this dependency. There is no opportunity for the client to compensate for a temporarily unavailable cache. For example, if you have a Varnish fleet that caches requests from an external REST service, then if that caching fleet goes down, from your service's perspective it’s as if the dependency itself went down. The other downside to an inline cache is that it needs to be built into the protocol or service it is caching for. If an inline cache for the protocol isn't available, then this inline caching isn't an option unless you want to build an integrated client or proxy service yourself.

Cache expiration

Some of the most challenging cache implementation details are picking the right cache size, expiration policy, and eviction policy. The expiration policy determines how long to retain an item in the cache. The most common policy uses an absolute time-based expiration (that is, it associates a time to live (TTL) with each object as it is loaded). The TTL is chosen based on client requirements, such as how tolerant the client can be to stale data, and how static the data is, because slowly changing data can be more aggressively cached. The ideal cache size is based on a model of the anticipated volume of requests and the distribution of cached objects across those requests. From that, we estimate a cache size can that ensures a high cache hit rate with these traffic patterns. The eviction policy controls how items are removed from the cache when it reaches capacity. The most common eviction policy is Least Recently Used (LRU).

So far, this is just a thought exercise. Real-world traffic patterns can differ from what we model, so we track the actual performance of our cache. Our preferred way to do this is to emit service metrics on cache hits and misses, total cache size, and number of requests to downstream services.

We have learned that we need to be deliberate about picking the cache size and expiration policy values. We want to avoid the situation where a developer arbitrarily picks some cache size and TTL values during initial implementation and then never goes back and validates their appropriateness at a later time. We have seen real-world examples of this lack of follow through leading to temporary service outages and exacerbation of ongoing outages.

Another pattern we use to improve resiliency when downstream services are unavailable is to use two TTLs: a soft TTL and a hard TTL. The client will attempt to refresh cached items based on the soft TTL, but if the downstream service is unavailable or otherwise doesn’t respond to the request, then the existing cache data will continue to be used until the hard TTL is reached. An example of this pattern is used in the AWS Identity and Access Management (IAM) client.

We also use the soft and hard TTL approach with backpressure to reduce the impact of downstream service brownouts. The downstream service can respond with a backpressure event when it is browning out, which signals that the calling service should use cached data until the hard TTL and only make requests for data that are not in its cache. We continue this until the downstream service removes the backpressure. This pattern allows the downstream service to recover from a brownout while maintaining availability of the upstream services.

Other considerations

An important consideration is what the cache behavior is when **errors** are received from the downstream service. One option for dealing with this is to reply to clients using the last cached good value, for example leveraging the soft TTL / hard TTL pattern described earlier. Another option we employ is to cache the error response (that is, we use a “negative cache”) using a different TTL than positive cache entries, and propagate the error to the client. The approach we choose in a given situation depends on the particulars of the service and by evaluating when it is better for clients to see stale data versus errors. Regardless of which approach we take, it is important that we ensure something is in the cache in error cases. If this is not the case and the downstream service is temporarily unavailable or otherwise unable to fulfill certain requests (for example when a downstream resource is deleted), the upstream service will continue to bombard it with traffic and potentially either cause an outage or exacerbate an existing one. We have seen real-world examples in which a failure to cache negative responses led to increased failure rates and faults.

Security is another important aspect of caching. When we introduce a cache to a service we evaluate and mitigate any additional security risks it introduces. For example, external caching fleets often lack encryption for serialized data and transport-level security. This is especially important if sensitive user information is retained in the cache. The issue can be mitigated by using something like Amazon ElastiCache for Redis, which supports in-transit and at-rest encryption. Caches are also susceptible to poisoning attacks, in which a vulnerability in the downstream protocol allows an attacker to populate a cache with a value under their control. This amplifies the impact of an attack, since all requests made while this value remains in the cache will see the malicious value. For one final example, caches are also susceptible to side-channel timing attacks. Cached values are returned faster than uncached values, so an attacker can use response time to gain information about requests that other clients or tenets are making.

One final consideration is the “thundering herd” situation, in which many clients make requests that need the same uncached downstream resource at approximately the same time. This can also occur when a server comes up and joins the fleet with an empty local cache. This results in a large number of requests from each server going to the downstream dependency, which can lead to throttling/brownout. To remedy this issue we use **request coalescing**, where the servers or external cache ensure that only one pending request is out for uncached resources. Some caching libraries provide support for request coalescing, and some external inline caches (such as Nginx or Varnish) do as well. In addition, request coalescing can be implemented on top of existing caches.

Amazon best practices and considerations

This article has touched on several Amazon best practices and the trade-offs and risks associated with caching. Here is a summary of the Amazon best practices and considerations that our teams use when they introduce a cache:

- Make sure there is a legitimate need for a cache that is justified in terms of cost, latency, and/or availability improvements. Ensure that the data is cacheable, which means that it can be used across multiple client requests. Be skeptical of the value a cache will bring, and carefully evaluate that the benefits will outweigh the added risks that the cache introduces.

- Plan to operate the cache with the same rigor and processes used for the rest of the service fleet and infrastructure. Don’t underestimate this effort. Emit metrics on cache utilization and hit rate to ensure the cache is tuned appropriately. Monitor key indicators (such as CPU and memory) to ensure that the external caching fleet is healthy and scaled appropriately. Set up alarms on these metrics. Make sure the caching fleet can be scaled up without downtime or mass cache invalidation (that is, validate that consistent hashing is working as expected.)

- Be deliberate and empirical in the choice of cache size, expiration policy, and eviction policy. Perform tests and use the metrics mentioned in the previous bullet to validate and tune these choices.
- Ensure that your service is resilient in the face of cache non-availability, which includes a variety of circumstances that lead to the inability to serve requests using cached data. These include cold starts, caching fleet outages, changes in traffic patterns, or extended downstream outages. In many cases, this could mean trading some of your availability to ensure that your servers and your dependent services don’t brown out (for example by shedding load, capping requests to dependent services, or serving stale data). Run load tests with caches disabled to validate this.
- Consider the security aspects of maintaining cached data, including encryption, transport security when communicating with an external caching fleet, and the impact of cache poisoning attacks and side-channel attacks.
- Design the storage format for cached objects to evolve over time (for example, use a version number) and write serialization code capable of reading older versions. Beware of poison pills in your cache serialization logic.
- Evaluate how the cache will handle downstream errors, and consider maintaining a negative cache with a distinct TTL. Don’t cause or amplify an outage by repeatedly asking for the same downstream resource and discarding the error responses.

Many service teams at Amazon use caching techniques. Despite the benefits of these techniques, we don’t take the decision to incorporate caching lightly because the downsides can often outweigh the upsides. We hope that this article helps you when you evaluate caching in your own services.