Caching.

Caching is a technique that stores copies of frequently used application data in a layer of smaller, faster memory in order to improve data retrieval times, throughput, and compute costs.

Caching can exist at any level of a system, from a single CPU to a distributed cluster. And the same fundamental design principles apply, regardless of where the cache is located. To determine if a system needs a cache, and what approaches get the best caching performance, you'll want to know the following caching fundamentals:

The purpose of caching

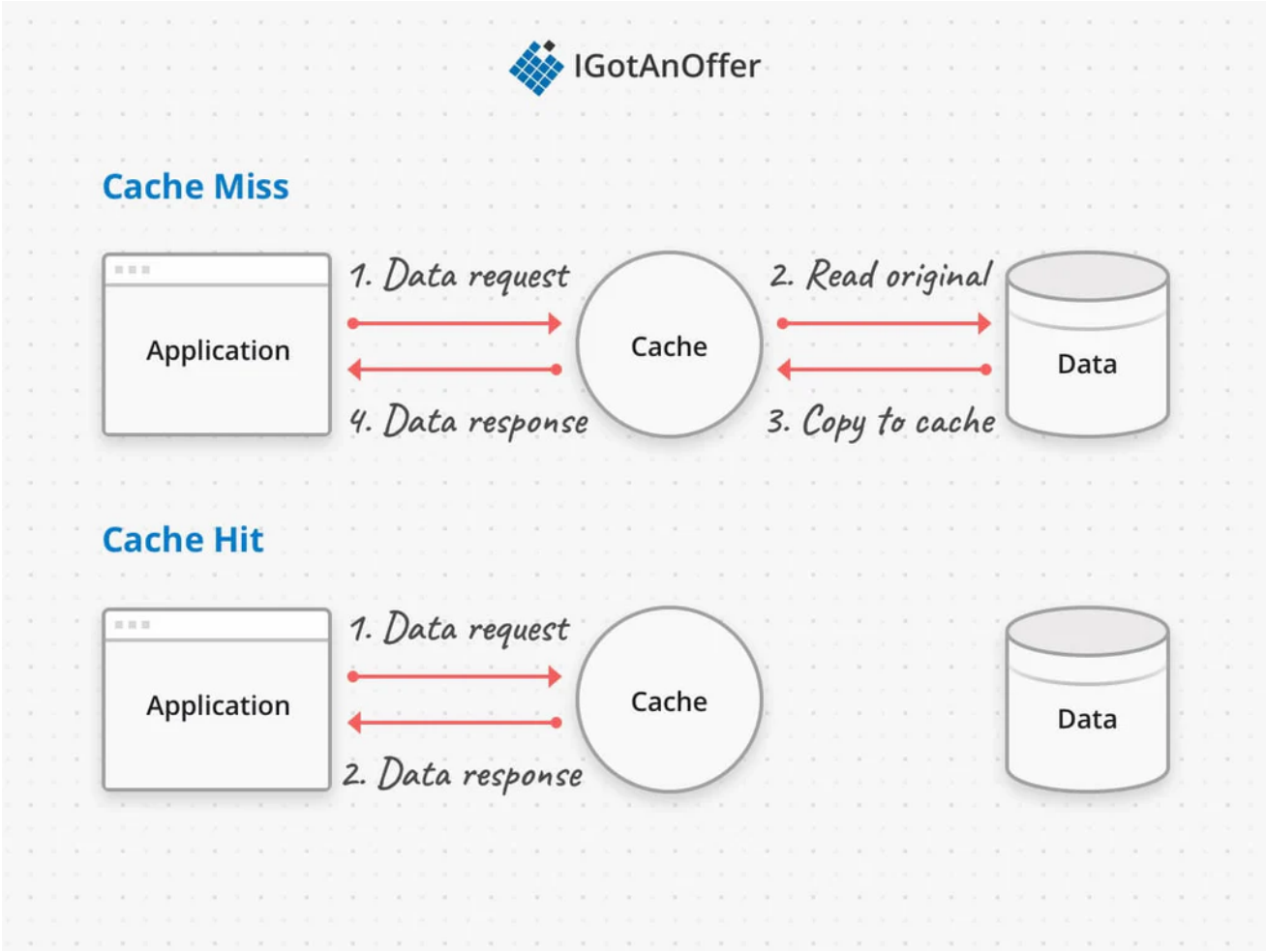
Caching is based on the **principle of locality**, which means that programs repeatedly access data located close to each other. There are two kinds of locality:

- **Temporal locality**, where data that has been referenced recently is likely to be referenced again (i.e. *time-based* locality).
-
- **Spatial locality**, where data that is stored near recently referenced data is also likely to be referenced (i.e. *space-based* locality).

We can speed up a system by making it faster to retrieve this locally relevant data. Caching does this by storing a "cached" copy of the data, in a smaller, faster data store. For example a CPU has a cache in specialized SRAM memory on the processor so it doesn't have to go all the way to RAM or Disk.

How a cache is used by an application

When a data request goes through a cache, a copy of the data is either already in the cache, which is called a **cache hit**, or it's not and the data has to be fetched from the primary store and copied into the cache, which is called a **cache miss**. A cache is performing well when there are many more cache hits than cache misses.



When to use caching

- Caching is most helpful when the data you need is slow to access, possibly because of slow hardware, having to go over the network, or complex computation.
- Caching is helpful in systems where there are many requests to static or slow to change data, because repeated requests to the cache will be up to date.
- Caching can also reduce load on primary data stores, which has the downstream effect of reducing service costs to reach performant response times.

Ultimately the features of a successful caching layer are highly situation-dependent. Creating and optimizing a cache requires tracking **latency** and **throughput** metrics, and tuning parameters to the particular data access patterns of the system.

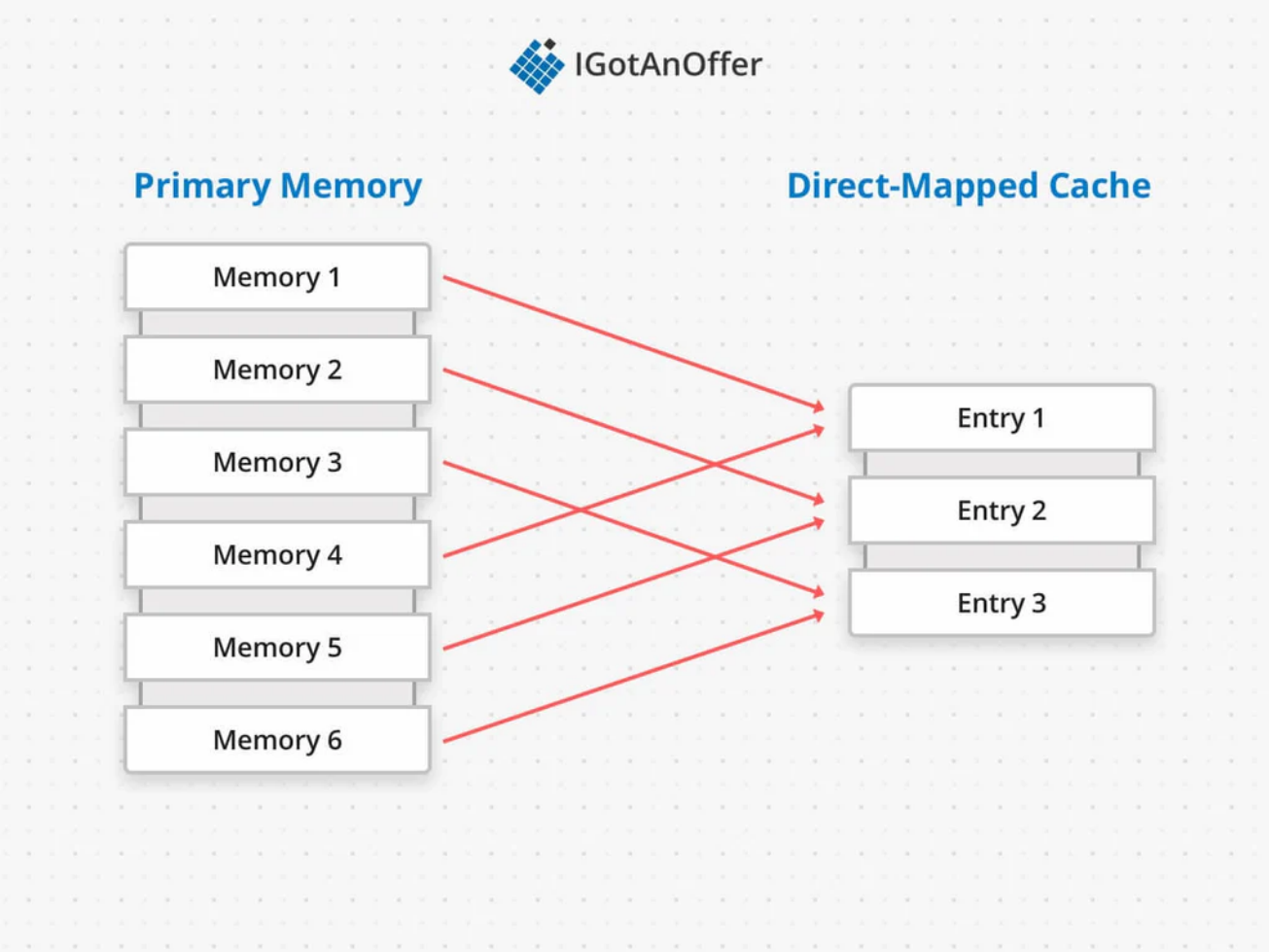
When not to use caching

- Caching isn't helpful when it takes just as long to access the cache as it does to access the primary data.
- Caching doesn't work as well when requests have low repetition (higher randomness), because caching performance comes from repeated memory access patterns.
- Caching isn't helpful when the data changes frequently, as the cached version gets out of sync and the primary data store must be accessed every time.

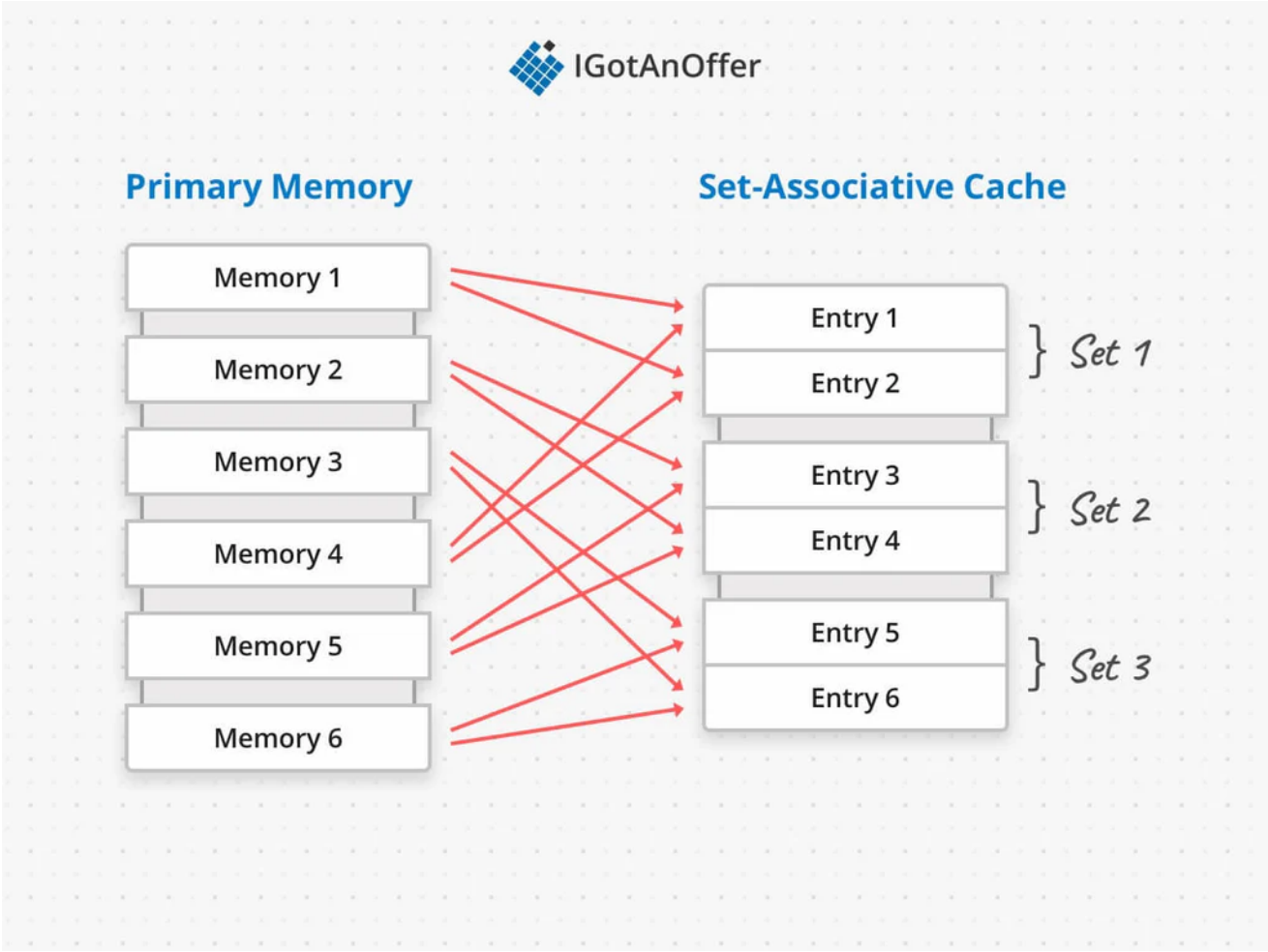
It's important to note that caches should not be used as permanent data storage - they are almost always implemented in volatile memory because it is faster, and should thus be considered transient. In section 2 below, we'll talk more about strategies for writing to a caching layer that don't cause data loss.

Cache performance

Cache performance is affected by the way data is mapped into the cache. Data is located the fastest in a cache if it's mapped to a single cache entry, which is called a **direct-mapped cache**. But if too many pieces of data are mapped to the same location, the resulting **contention** increases the number of cache misses because relevant data is replaced too soon.



The widely accepted solution is to use a **set-associative cache** design. Each piece of data is mapped to a set of cache entries, which are all checked to determine a cache hit or miss. This is slightly slower than only having to check a single entry, but gives flexibility around choosing what data to keep in the cache. Below, we'll talk more about cache replacement algorithms and their design tradeoffs.



One final thing to note is the performance of a cache on startup. Since caching improves repeated data access, there's no performance benefit on a **cold start**. A cache can be **warmed up** to make sure it doesn't lag on startup by **seeding** it with the right data before any operations occur. This can be a set of static data that is known beforehand to be relevant, or predicted based on previous usage patterns.

Now that we've covered the basics of caching, let's look more closely at different ways of implementing a cache. First, we'll look at "policies" for writing to a cache.

Cache writing policies

There are interesting design tradeoffs between the speed of writing and the consistency with persisted data. A cache is made of copies of data, and is thus transient storage, so when writing we need to decide when to write to the cache and when to write to the primary data store.

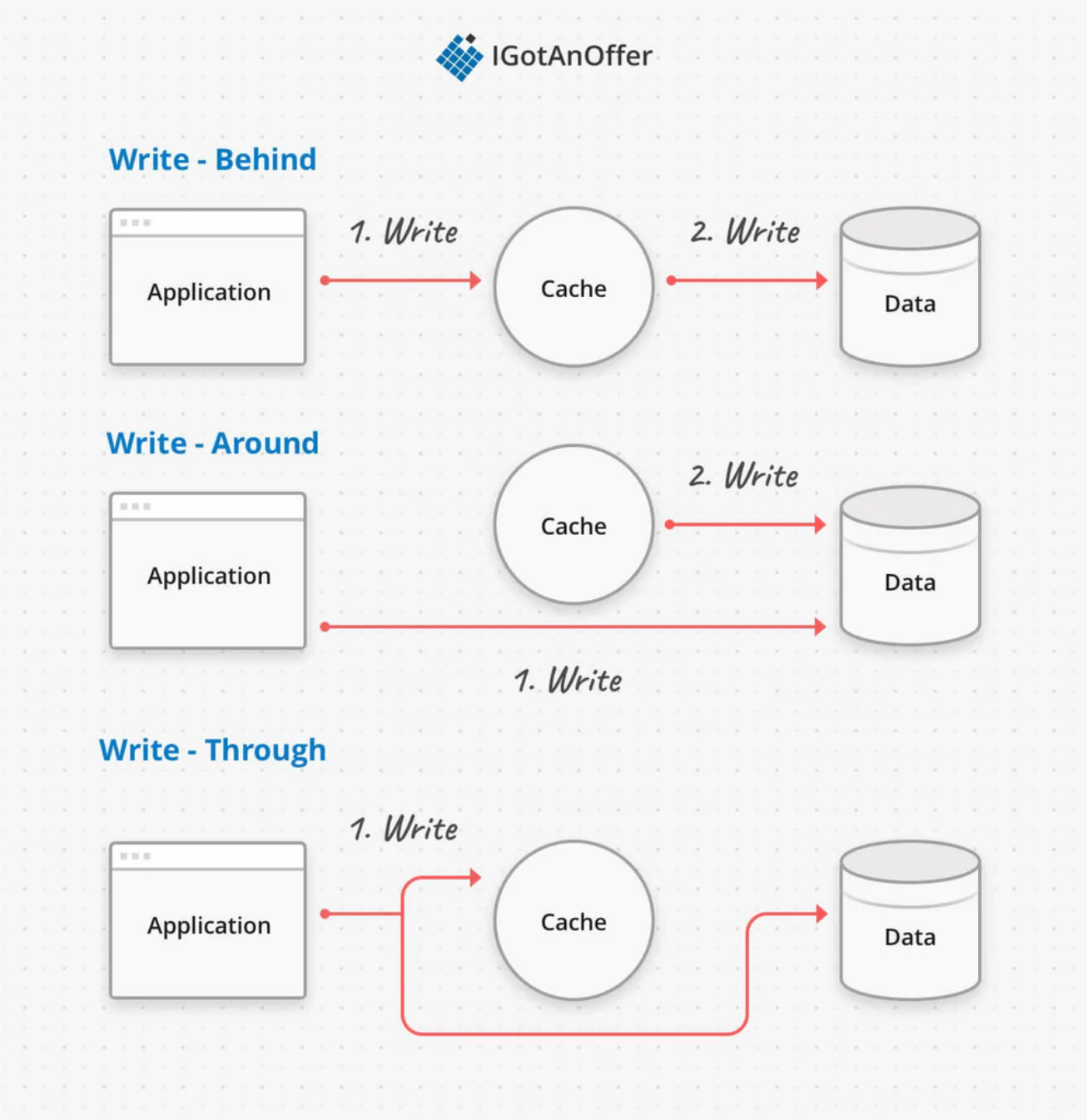
A **write-behind cache** writes first to the cache, and then to the primary datastore. This can either mean that the primary data store is updated almost immediately, or that the data is not persisted until the cache entry is replaced, in which case it's tagged with a **dirty bit** to keep track that the data is out of sync.

It is possible that at the moment a write-behind cache fails that some data has not been persisted, and is lost. Ultimately writes will be fastest in a write-behind cache, and it can be a reasonable choice if the cached data tolerates possible data loss.

In a **write-around cache** design, the application writes directly to the primary datastore, and the cache checks with the primary datastore to keep the cached data valid. If the application is accessing the newest

data, the cache might be behind, but the write doesn't have to wait on two systems being updated and the primary datastore is always up to date.

Finally, a **write-through cache** updates both the cache data and the backing data at the same time. If the cache layer fails, then the update isn't lost because it's been persisted. In exchange, the write takes longer to succeed because it needs to update the slower memory.



A cache has a limited amount of available memory, so at some point we need to clear out the old data and make space for more relevant data. The choice of which data to remove is made by a cache replacement policy.

Cache replacement policies

The **cache replacement policy** is an essential part of the success of a caching layer. The replacement policy (also called **eviction policy**) decides what memory to free when the cache is full.

A good replacement policy will ensure that the cached data is as relevant as possible to the application, that is, it utilizes the principle of locality to optimize for cache hits. Replacement policies are fine-tuned to particular use cases, so there are many different algorithms and implementations to choose from. But they're all based on a few fundamental policies:

LRU - least recently used

In an LRU replacement policy, the entry in the cache that is the oldest will be freed. LRU performs well and is fairly simple to understand, so it's a good default choice of replacement policy.

To implement LRU the cache keeps track of recency with **aging bits** that need to get updated on every entry every time data is accessed. Although LRU makes efficient decisions about what data to remove, the computational overhead of keeping track of the aging bits leads to approximations like the clock replacement policy.

A clock replacement policy approximates LRU with a clock pointer that cycles sequentially through the cache looking for an available entry. As the pointer passes an in-use entry, it marks the **stale bit**, which gets reset whenever the entry is accessed again. If the clock pointer ever finds an entry with a positive stale bit, it means the cached data hasn't been accessed for an entire cycle of the clock, so the entry is freed and used as the next available entry.

LRU-based algorithms are well suited for applications where the oldest data are the least likely to be used again. For example, a local news outlet where users mostly access today's news could use a CDN with LRU replacement to make today's news faster to load. After that day has passed, it becomes less likely that the news article is accessed again, so it can be freed from the cache to make space for the next day's news.

LFU - least frequently used

In an LFU replacement policy, the entry in the cache that has been used the least frequently will be freed. Frequency is tracked with a simple access count in the entry metadata.

LFU replacement policies are especially helpful for applications where infrequently accessed data is the least likely to be used again. For example, an encyclopedia-like service could have certain articles that are popular (e.g. about celebrities) and others that are more obscure.

With a CDN running LFU replacement would mean the popular articles are persisted in the cache and faster to load, while the obscure articles are freed quickly.

Balancing LRU and LFU

Both LRU and LFU have advantages for particular data access patterns, so it's common to see them combined in various ways to optimize performance. An LFRU (Least Frequently Recently Used) replacement policy is one such example. It takes into account both recency and frequency by starting with LFU, and then moving to LRU if the data is used frequently enough.

To respond to *changing* data access patterns, an ARC (Adaptive Replacement Cache) takes the LFRU approach and then dynamically adjusts the amount of LFU and LRU cache space based on recent replacement events. In practice, ARC will outperform LFU and LRU, but requires the system to tolerate much higher complexity which isn't always feasible.

ARC's are valuable when access patterns vary, for example a search engine where sometimes requests are a stable set of popular websites, and sometimes requests cluster around particular news events.

Expiration policies

In distributed systems there is often an explicit **expiration policy** or **retention policy** for making sure there is always space in the cache for new data. This parameter specifies an amount of time after which a resource is freed if it hasn't been used, called the **Time To Live (TTL)**.

Finding the right TTL is one way to optimize a caching layer. Sometimes explicit removal policies are **event-driven**, for example freeing an entry when it is written to.

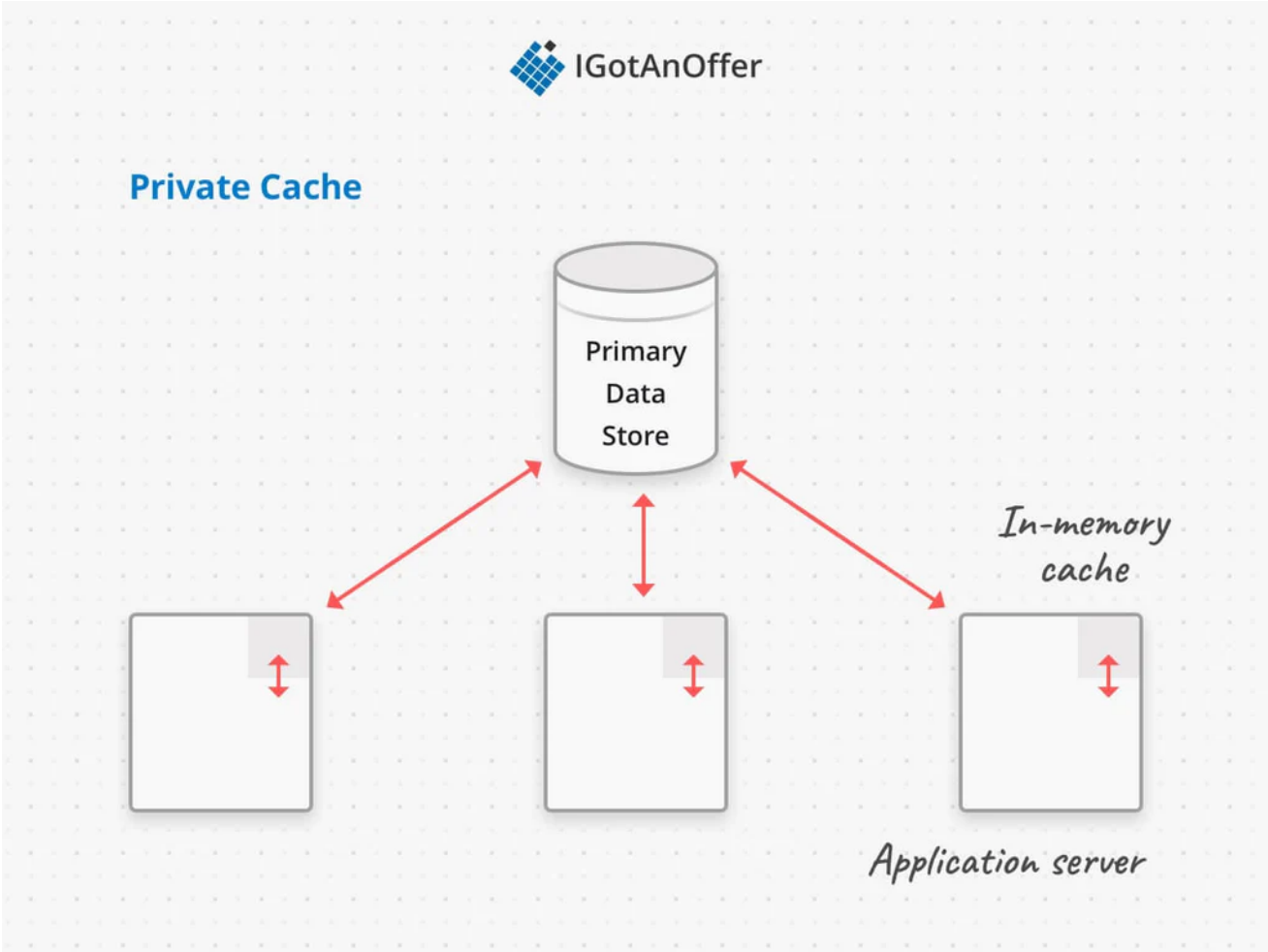
```
A self note we should always have the thought to predict the most used/frequent data by
predicting what most of the clients wants this wat we can build a perfect cache system.....
```

Distributed cache

In a distributed system a caching layer can reside on each machine in the application service cluster, or it can be implemented in a cluster isolated from the application service.

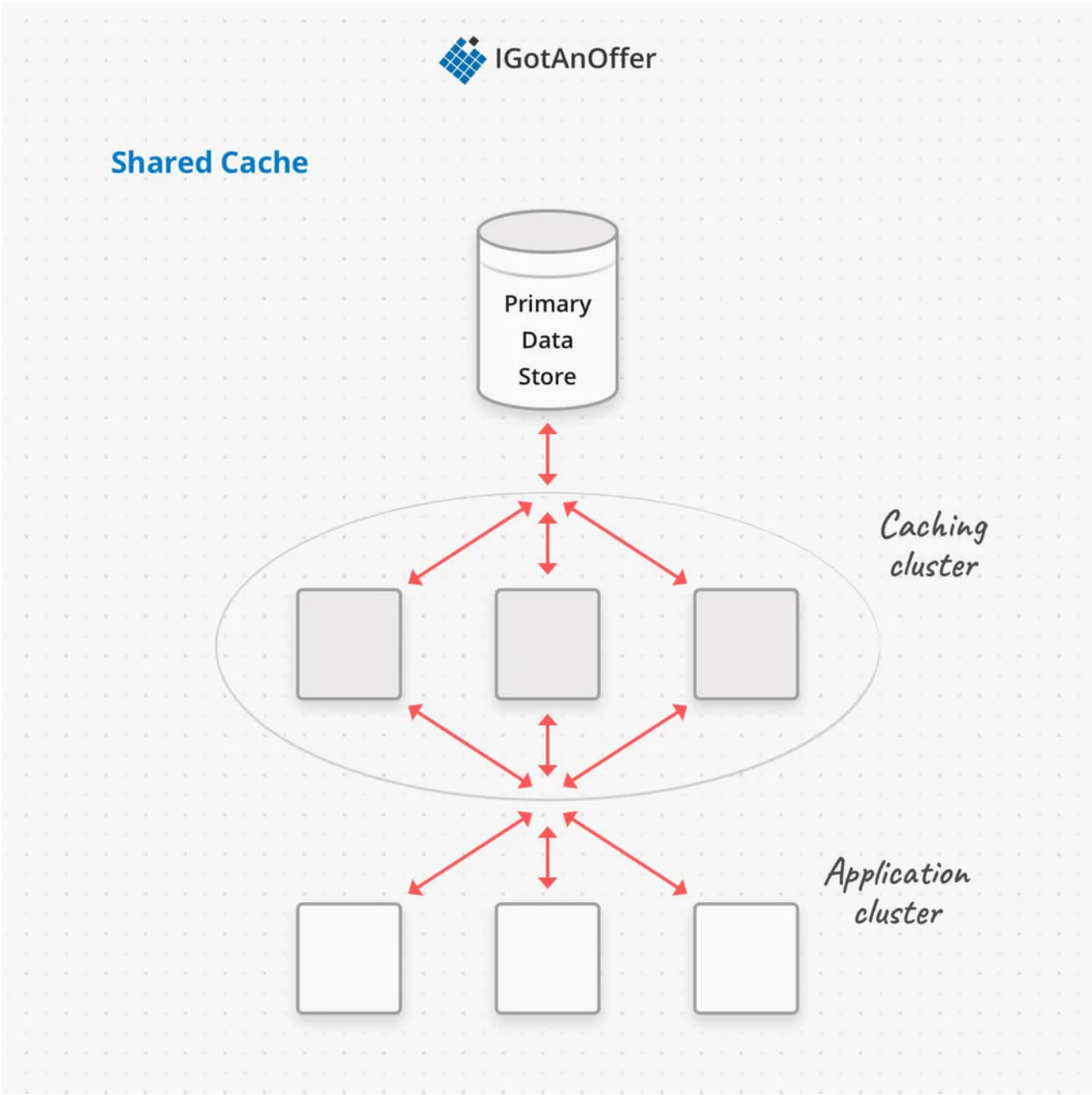
A **private** cache exists in the memory of each application computer in the service cluster. Since each machine stores its own cache, the cached data will get out of sync based on what requests have been directed to that machine, and at what time.

The main advantage of in-memory distributed caches is speed - since the cache is in memory, it's going to function a lot faster than a shared caching layer that requires network requests to operate. **Redis** and **Memcached** are two common choices for private cache implementations.



A **shared** cache exists in an isolated layer. In fact, the application might not be aware that there's a cache. This allows the cache layer and application layer to be scaled independently, and for many different types of application services to use the same cache.

The application layer needs to be able to detect the availability of the shared cache, and to switch over to the primary data store if the shared cache ever goes down. A shared cache implemented on its own cluster has the same resiliency considerations as any distributed system - including failovers, recovery, partitioning, rebalancing, and concurrency.



Private and shared caches can be used together, just like CPU architectures have different levels of caching. The private cache can also be a fallback that retains the advantages of caching in case the shared cache layer fails.

IGotAnOffer

Examples of Caching

Hardware	Services	Applications
CPUs	API servers	Browser sessions
GPUs	CDN	Web artifacts (HTML, JS, CSS, images)
TLBs	DNS	Simulations
	Databases	Scientific computations
	Search engines	