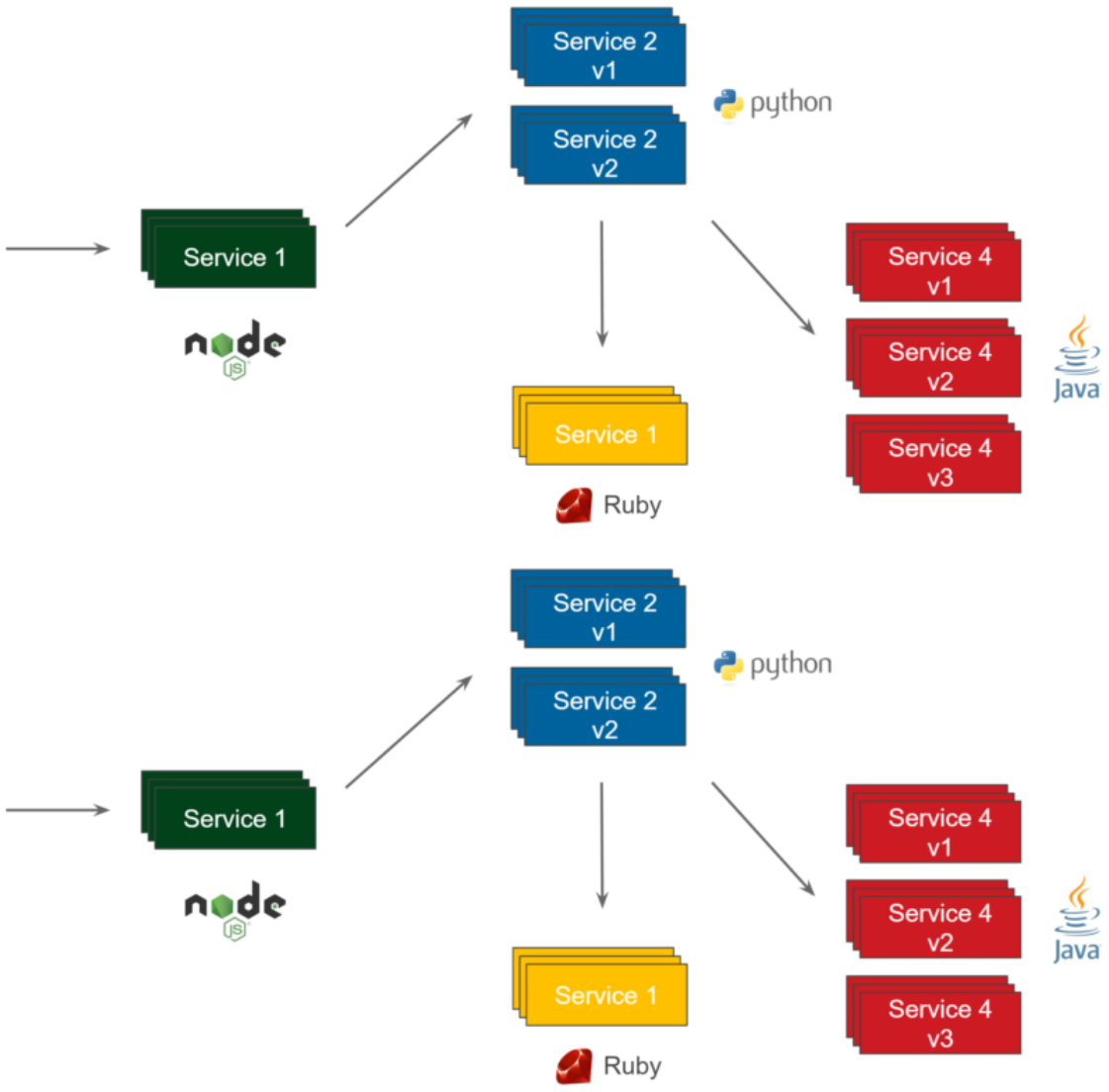
A

Where Is My Cache? Architectural Patterns for Caching Microservices

Rafal Leszko | Sep 10, 2019

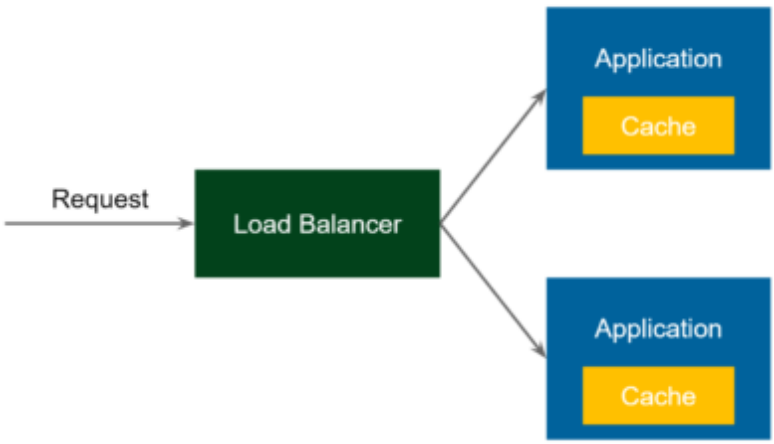
I'm sure you use caching somewhere in your system. This can be either to improve performance, reduce backend load, or to decrease downtime. Everybody uses caching. Caching is everywhere. However, in which part of your system should it be placed? If you look at the following diagram representing a simple microservice architecture, where would you draw the "cache" rectangle for your microservice cache strategy?



A diagram representing a simple microservice architecture.

There is more than one right answer to this question. You could place the cache inside each service or as a completely separate cache server. You could also put it in front of each service, or even as a sidecar container which belongs to the service—there are many options. I've already mentioned some of them in the Hazelcast Sidecar Container Pattern blog post. Here, let's summarize all the options you have in the microservice world and describe **Caching Architectural Patterns**.

Pattern 1: Embedded Cache



Example of an embedded cache architecture.

The simplest possible caching pattern is **Embedded Cache**. In the diagram above, the flow is as follows:

1. **Request** comes in to the **Load Balancer**
2. **Load Balancer** forwards the request to one of the **Application** services
3. **Application** receives the request and checks if the same request was already executed (and stored in cache)
   - If yes, then return the cached value
   - If not, then perform the long-lasting business operation, store the result in the cache, and return the result

This long-lasting business operation can be anything worth caching; for example, performing a computation, querying a database, or

calling an external web service.

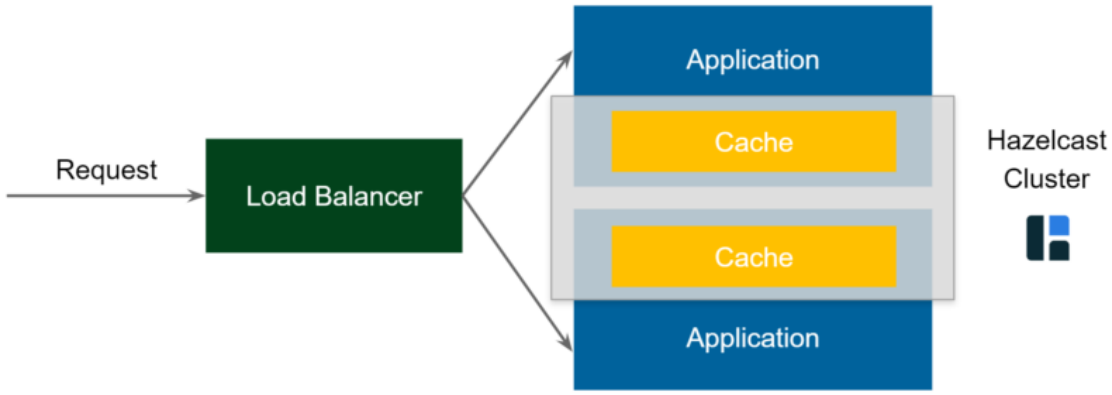
Such caching logic is very simple and we could quickly write code for it, either using built-in data structures or some caching libraries (like Guava cache). We could also place the cache in the application layer and use the caching capabilities provided by most web frameworks. For example, in the case of Spring, adding a caching layer requires nothing more than adding the `@Cacheable` annotation to the method.

```
@Service
public class BookService {
    @Cacheable("books")
    public String getBookNameByIsbn(String isbn) {
        return findBookInSlowSource(isbn);
    }
}
```

With this implementation, every time the `getBookNameByIsbn(String isbn)` method is called, Spring first checks if this method was already executed with the given `isbn` parameter. If yes, then the cached value is returned instead of executing the `findBookInSlowSource(String isbn)` method. You can find the complete code for this example here.

There is one issue with the embedded cache approach. Imagine that there is a request made to our system, and the first time it's forwarded to the application service on the top. Then, the same request comes in, but this time the load balancer forwards it to the application service on the bottom. Now, we received the same request twice, but had to perform the business logic twice, since the two caches on the diagram are completed separately. Let's improve the pattern and use a distributed cache library.

Pattern 1*: Embedded Distributed Cache



An embedded distributed cache using Hazelcast.

Embedded Distributed Cache is still the same pattern as **Embedded Cache**; however, this time we'll use Hazelcast instead of the default non-distributed cache library. From now on, all caches (embedded in all applications) form one distributed caching cluster. Since Hazelcast is written in Java, you can use it together with Spring; all you need to do is add the following `CacheManager` configuration.

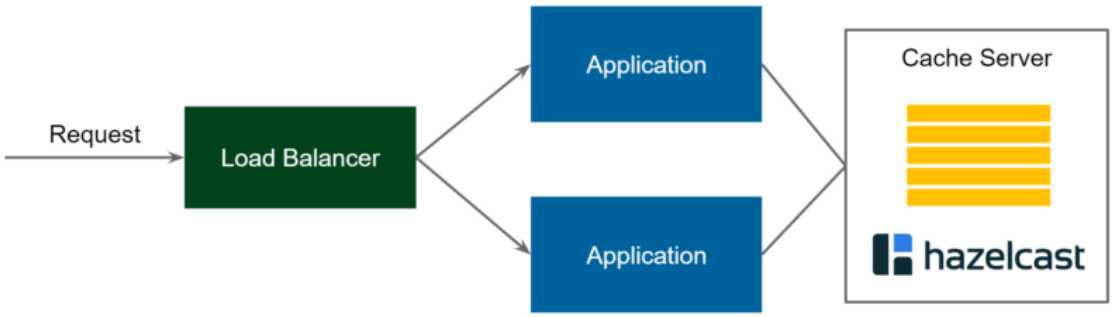
```
@Configuration
public class CacheConfiguration {
    @Bean
    CacheManager cacheManager() {
        return new
HazelcastCacheManager(Hazelcast.newHazelcastInstan
ce());
    }
}
```

With these few lines of code, we made Spring use Hazelcast for all of the caching functionalities it provides. You can check the complete implementation of this example here.

Using an embedded cache (both distributed and non-distributed) is simple because it does not require any additional configuration or deployment. What's more, you always get the low-latency data transfer, because the cache runs physically in the same JVM. We'll

have a closer look at the pros and cons of this solution later. Now let's now move on to a completely different caching pattern, **Client-Server**.

Pattern 2: Client-Server Cache



A client-server cache configuration.

This time, the flow presented on the diagram is as follows:

1. **Request** comes into the **Load Balancer** and is forwarded to one of the **Application** services
2. **Application** uses cache client to connect to **Cache Server**
3. If there is no value found, then perform the usual business logic, cache the value, and return the response

This architecture looks similar to the classic database architecture. We have a central server (or more precisely a cluster of servers) and applications connect to that server. If we were to compare **Client-Server** pattern with **Embedded Cache**, there are two main differences:

- The first one is that the Cache Server is a separate unit in our architecture, which means that we can manage it separately (scale up/down, backups, security). However, it also means that it usually requires a separate Ops effort (or even a separate Ops team).
- The second difference is that the application uses a cache client library to communicate with the cache and that means that we're no longer limited to JVM-based languages. There is a well-defined protocol, and the programming language of the server part can be different than the client part. That is actually one of the reasons

why many caching solutions, such as Redis or Memcached, offer only this pattern for their deployments.

If you're interested in seeing this pattern in action, then the simplest way is to start a Hazelcast Cache server on Kubernetes using Hazelcast Helm Chart with the following command:

```
$ helm install hazelcast/hazelcast
```

Then, you can use the following example to run the client application. The most interesting part is the Spring cache configuration, which looks as follows:

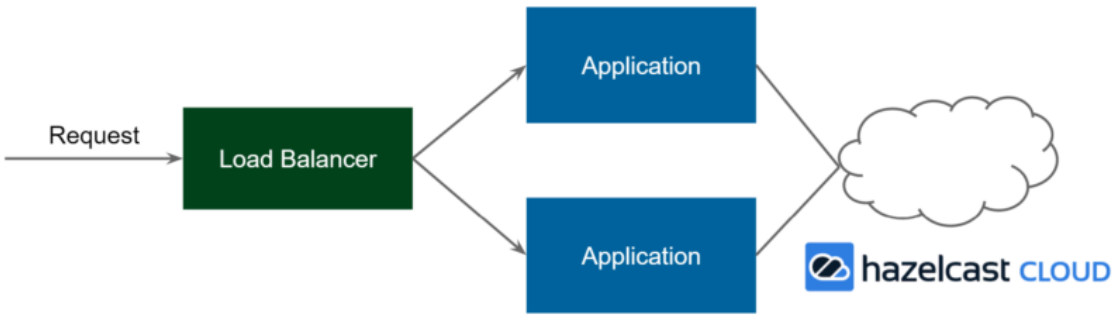
```
@Configuration
public class CacheConfiguration {
    @Bean
    CacheManager cacheManager() {
        ClientConfig clientConfig = new
ClientConfig();

clientConfig.getNetworkConfig().getKubernetesConfi
g()
            .setEnabled(true);
        return new
HazelcastCacheManager(HazelcastClient

.newHazelcastClient(clientConfig));
    }
}
```

As you can see, all we have to configure is to specify the Hazelcast client configuration; all the rest is done automatically.

I mentioned before that the first difference between an **Embedded** cache and a **Client-Server** cache is that the former is managed

separately. A separate Ops team can even manage it, or you could go even further and move the management part into the **Cloud**.

Pattern 2*: Cloud Cache



Cloud cache configuration using Hazelcast Cloud.

In terms of the architecture, **Cloud** is like **Client-Server**, with the difference being that the server part is moved outside of your organization and is managed by your cloud provider, so you don't have to worry about all of the organizational matters.

If you are interested in an example, you can create a Hazelcast cluster on the **Hazelcast Cloud** platform at: https://cloud.hazelcast.com/. Then, you can find a complete client application here. The most interesting part is the Spring configuration:

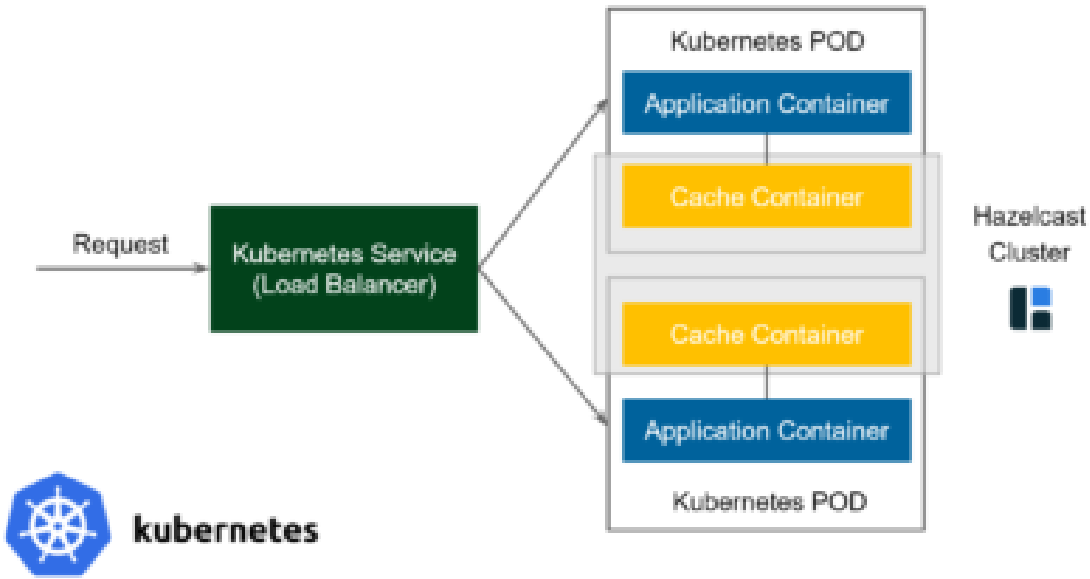
```
@Bean
CacheManager cacheManager() {
    ClientConfig clientConfig = new
ClientConfig();

clientConfig.getNetworkConfig().getCloudConfig()
        .setEnabled(true)

.setDiscoveryToken("KSXFDTi5HXPJGR0wRAjLgKe45tvEEh
d");
    clientConfig.setGroupConfig(new
GroupConfig("test-cluster", "b2f9845"));
```

```
    return new HazelcastCacheManager(

HazelcastClient.newHazelcastClient(clientConfig));
}
```

Using the **Client-Server** pattern is simple, and using the **Cloud** pattern is even simpler. Both of them bring similar benefits like separating cache data from the application, separate management (scaling up/down, backups), and the possibility to use any programming language. Nevertheless, one thing becomes more difficult—the **latency**. In the case of the **Embedded** pattern, the cache was always located on the same machine (even in the same JVM) as the application. However, when the server part is detached, we now need to think about its physical location. The best option is to use the same local network (or the same VPC in case of Cloud solutions).

The first two architectural patterns, **Embedded** and **Client-Server**, may sound obvious because these ideas are quite old. Now, let's move to a new and slightly more unusual pattern, the **Cache as a Sidecar**.

Pattern 3: Sidecar Cache



The diagram above is Kubernetes-specific, because the **Sidecar** pattern is mostly seen in (but not limited to) Kubernetes environments. In Kubernetes, a deployment unit is called a POD. This POD contains

one or more containers which are always deployed on the same physical machine. Usually, a POD contains only one container with the application itself. However, in some cases, you can include not only the application container but some additional containers which provide additional functionalities. These containers are called sidecar containers.

This time, the flow looks as follows:

1. **Request** comes to the **Kubernetes Service** (Load Balancer) and is forwarded to one of the **PODs**
2. **Request** comes to the **Application Container** and **Application** uses the cache client to connect to the **Cache Container** (technically Cache Server is always available at `localhost`)

This solution is a mixture of the **Embedded** and **Client-Server** patterns. It's similar to **Embedded Cache**, because:

- Cache is always at the same machine as the application (low latency)
- Resource pool and management activities are shared between cache and application
- Cache cluster discovery is not an issue (it's always available at `localhost`)

It's also similar to the **Client-Server** pattern, because:

- Application can be written in any programming language (it uses the cache client library for communication)
- There is some isolation of cache and application

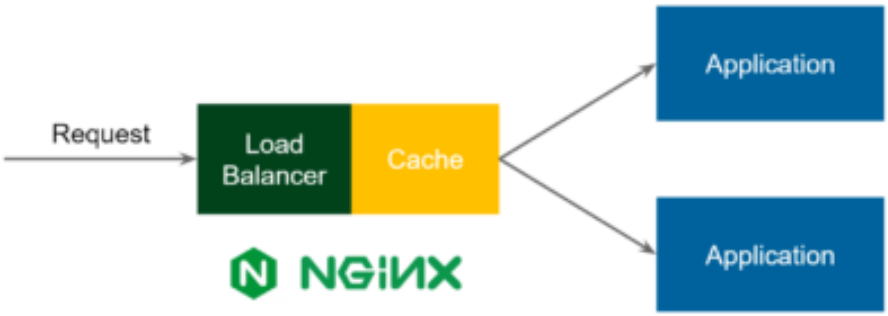
This time, the Spring configuration would look as follows:

```
@Bean
CacheManager cacheManager() {
    ClientConfig clientConfig = new
```

```
ClientConfig();

clientConfig.getNetworkConfig().addAddress("localh
ost:5701");
    return new
HazelcastCacheManager(HazelcastClient

.newHazelcastClient(clientConfig));
}
```

If you're interested in more details of how Hazelcast works as a sidecar container, please read the Hazelcast Sidecar Container Pattern blog. Now let's discuss a completely different pattern, the **Reverse Proxy**.

Pattern 4: Reverse Proxy Cache



So far, in each scenario, the application was aware that it uses a cache. This time, however, we put the caching part in front of the application, so the flow looks as follows:

1. **Request** comes in to the **Load Balancer**
2. **Load Balancer** checks if such a request is already cached
3. If yes, then the response is sent back and the **Request** is not forwarded to the **Application**

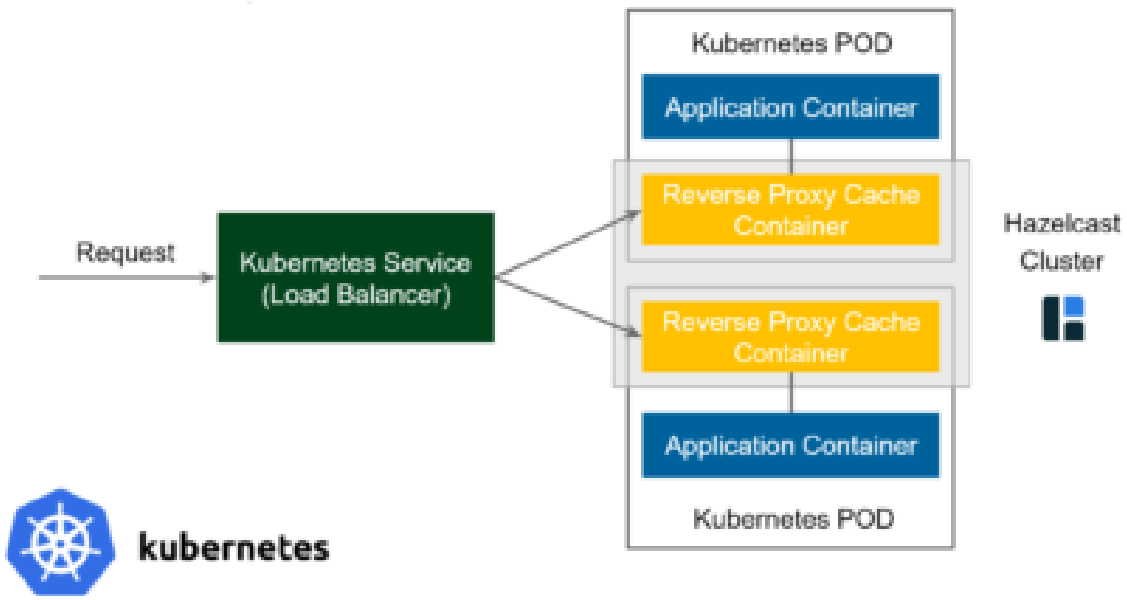
Such a caching solution is based on the protocol level, so in most cases, it's based on HTTP, which has some good and bad implications:

- The good thing is that you can specify the caching layer as a configuration, so you don't need to change any code in your application.
- The bad thing is that you cannot use any application-based code to invalidate the cache, so the invalidation must be based on timeouts (and standard HTTP TTL, ETag, etc.).

NGINX provides a mature reverse proxy caching solution; however, data kept in its cache is not distributed, not highly available, and the data is stored on disk.

One improvement we could do to the **Reverse Proxy** pattern is to inject the HTTP Reverse Proxy as a sidecar. Here's how you can do it:

Pattern 4*: Reverse Proxy Sidecar Cache

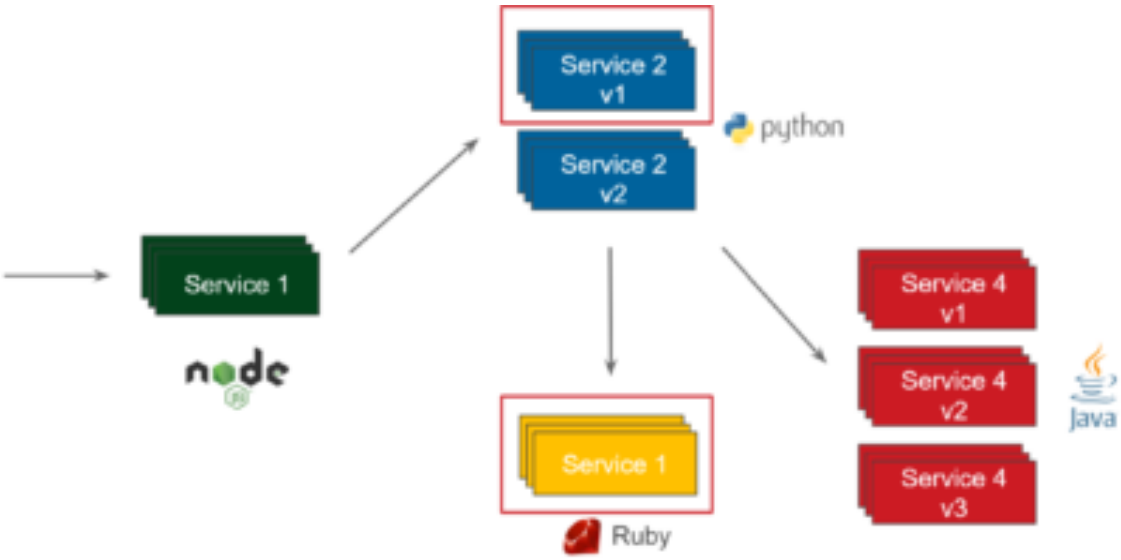


Again, when it comes to the sidecar, the diagram is limited to the Kubernetes environment. The flow looks as follows:

1. **Request** comes in to the **Kubernetes Service** (Load Balancer) and is forwarded to one of the **PODs**
2. Inside the **POD**, it's the **Reverse Proxy Cache Container** (not **Application Container**) that receives the **Request**
3. **Reverse Proxy Cache Container** checks if such a request is already cached

4. If yes, then it sends the cached response (and does not even forward the **Request** to **Application Container**)

Application Container is not even aware that the cache exists. Think about the microservice system presented at the beginning of this blog post. With this pattern, we could look at the whole system and specify (in the Kubernetes configuration file) that `Service 2 v1` and `Service 1` should be cached.



The whole idea of changing the system in a declarative manner (without modifying the source code of each service) is very similar to the concept of **Istio** (and **Service Mesh** in general). Nevertheless, at the time of this writing, **Istio** does not provide caching functionality. What's more, at the moment, there is no mature **HTTP Reverse Proxy Cache Sidecar** solution at all. However, I believe it will become more and more popular since a few projects are already actively working on some stable implementations. If you're interested in the experimental version, check out [Hazelcast PoC of HTTP Reverse Proxy Sidecar caching](#).

Pros & Cons

We mentioned a lot of caching patterns that you can use in your microservice system. Here's a handy list of the pros and cons in one table.

	Pros	Cons
Embedded (Distributed) Cache	Simple configuration and deployment	Not flexible management (scaling, backup)
	Low-latency data access	Limited to JVM-based applications
	No separate Ops effort needed	Data collocated with applications
Client-Server (Cloud) Cache	Data separate from applications	Separate Ops effort
	Separate management (scaling, backup)	Higher latency
	Programming-language agnostic	Server network requires adjustment (same region, same VPC)
Sidecar Cache	Simple configuration	Limited to container-based environments
	Programming-language agnostic	Not flexible management (scaling, backup)
	Low latency	Data collocated with application PODs
	Some isolation of data and applications	
Reverse Proxy (Sidecar) Cache	Configuration-based (no need to change applications)	Difficult cache invalidation
	Programming-language agnostic	No mature solutions yet
		Protocol-based (e.g., works only with HTTP)

> Consistent with containers and the microservice world

Conclusion

In the microservice world, there are many ways in which you can configure the cache in your system. As a rule of thumb, you should use caching only in one place. That means that you should never combine the patterns and cache in multiple layers at the same time (such as in both the HTTP and the application levels). Such an approach could lead to even more problems with cache invalidation and would make your system error-prone and difficult to debug. If you use caching in one place, then it's your choice as to which pattern to use. The most conservative approach is the good old **Client-Server** (or **Cloud**) pattern; however, I believe that the future lies in the declarative-style **HTTP Reverse Proxy Cache** injection. You will probably hear about it more and more in the coming months.