Grpc Caching service

# Introduction

Caching is integral part of Cloud native applications as these applications stores state externally. Apart, from that there are many business applications that uses redis as a cache to gain performance and also reduce the cost of accessing the database.

Most popular product that is used for caching is Redis. Redis is a great product and battle tested key value store that offers excellent performance with wide availability. The other products that can be used are cloud provider specific databases ( COSMOS DB, AWS Document DB), SQL server or any other database server and Mongo DB.

# Redis

Redis is a world class cache storage. But it is not without its own set of problems.

## Capability mismatch

Redis is provided as a packaged service by various cloud providers like Azure, AWS, and PCF. These packaged versions could differ in the capability that they offer. For example, the packed version of Redis provided by PCF disables LUA scripting which could cause issues with applications that uses LUA scripts.

## Boundary crossing - Performance

Boundary crossing comes with extra performance overhead. When an application that needs Redis to enhance to performance is deployed on EKS or AKS cluster then this application will have chatty interaction with Redis. Such chatty cross boundary calls are performance degraders.

## Additional security configurations

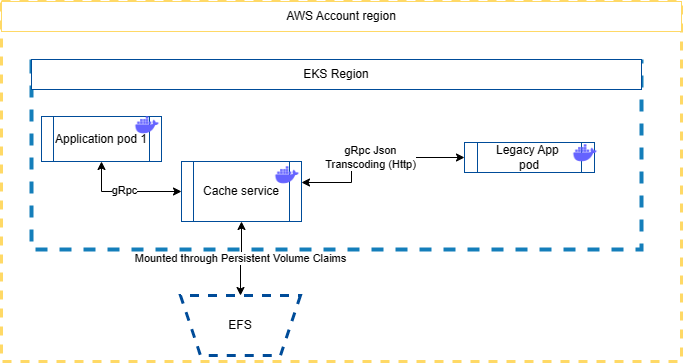
This is a paramount concern and is an on-time configuration. To have defense in depth the interaction between backing service and the application must be secured using TLS (SSL). Thus, the benefits of terminating a TLS at load balancer level and making internal calls as unsecured gets overridden. The other option for this is to have Redis also deployed as a pod in EKS. But that would become an overhead for the platform supporting teams or dev teams that they keep these instances of Redis or caching persistence to latest versions. Additional CVE from Product and their updates also should be owned by application team and owing to their focus and priority difference a potential security loop could be left open.

All these aspects are equally applicable to any open-source product that is used for caching.

# Grpc Caching service

## Architecture diagram

The diagram below assumes that the cache service is deployed in AWS cluster. In case of Azure or any other cluster the file volumes and respective Kubernetes service and terminology would vary.



This idea stems from the fact that when using EFS or Azure disk the performance of caching is increased as there is no boundary caching. Also needs for additional security needs discussed above does not arise. The salient concerns with respect to custom caching are addressed as below

## Scalability

The application is horizontally scalable with multiple pods mounting to same caching volume (EFS or Azure disk). There is no loss of functionality as each pods implement a file system watcher that loads the delta changes to caches on the disk.

## Performance

This is a key aspect of any caching service. Hence, a deep dwelling is required

### Framework level choices

.NET 7 which is one of the fastest frameworks has been chosen. This framework is also cloud native and hence brings the best practices of the cloud with it. It uses a RHEL (not windows) image for deployment. Another reason why .NET 7 is chosen is that, In .NET 7 gRPC JSON Transcoding is enabled for unary or binary communication in non-streaming usecase.

### Application library choices

This application uses [Net.DistributedFileStoreCache](https://github.com/JonPSmith/Net.DistributedFileStoreCache). The performance benchmarks of it can be seen in the link.

### gRPC protocol

what is a gRpc protocol can be seen at [gRPC | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc). Since gRPC is being used the performance benefits and the goodness of gRPC is fully harnessed

## No Boundary crossing

There is no need to have boundary crossing with this service. This can be deployed with business apps in the same cluster with cloud provider specific file system volume mount.

## TLS security

The current version is mostly going to be used as ClusterIP deployment (in cluster access only). In case there is any specific reason for doing TLS deployment for external exposure, it can be provided as feature list to our team, and it can be built. In gRpc transcoding usecase, when the same port is used for Http1 and Http2 multiplexed request HTTPS must be available till the hosting process (Kestrel). The TLS termination cannot happen at cluster level. To avoid that this service uses two ports, one for gRPC and one for JSON transcoding.

## Retrofitting and SDK

The communication to the service is via gRpc. To safeguard app developers from nuances of gRpc a .NET SDK library is provided. If there are other language frameworks that can work with gRpc, the proto files can be shared by to those teams, and they can build their language specific framework. If there are another old framework that cannot support gRpc then gRpc json transcoding will be used. Swagger is enabled gRpc services. The sidecar will then communicate with the service using gRPc.

## Load Balancing

One of the main issues with the gRPC protocol is its load balancing using the Kubernetes service. This is not an issue when using gRPC trans coding as in that case only Http/1.1 calls are made and they can be easily handled by Kubernetes services. The real issue comes when the Kubernetes service has to load balance between the Kubernetes pods that are exposed via gRpc. Since gRpc uses Http/2.0 protocol that makes a long duration connection with the server, the traditional method of load balancing by the Kubernetes Service across multiple pods get failed. To circumvent this issue, the following are done.

1. Make the Kubernetes ClusterIP service as headless. This is done by setting ClusterIp to ‘none’.
2. In the grpc client add a DNS resolver and set the time period for grpc refresh interval to say once a minute. This is important because if the pod gets moved to some other node as a part of Kubernetes Failover strategy, then the a DNS probe has to happen to get the new IP.
3. Then the most crucial part is the actual address of the service should be of the scheme “dns:///” and not http or https.
4. Setup the service configuration strategy of the line to have a round robin configuration or any other suitable strategy.
5. The pod must listen in port 80 only.

The main reason of using an headless service is that when DNS lookup is instantiated, it would result in providing the POD IP that are load balanced by it. These pod IP addresses are then load balanced by the gRPC client. Here we are using the gRPC client load balancing strategy.