Reducing Execution Time in FaaS Cloud Platforms

Ivan Rosa[a], Filipe Freitas[a], José Simão[a,b,c]
[a]Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa
[b]INESC-ID Lisboa
[c]Future Internet Technologies (FIT-ISEL)
filipe.freitas@isel.pt jose.simao@isel.pt

Abstract— **This article research into the significance of caching within Function-as-a-Service (FaaS) environments, exploring how caching strategies can substantially enhance performance and scalability in the realm of serverless computing. A versatile caching architecture for FaaS is introduced, tailored to accommodate different caching strategies. The architecture is implemented by extending an open-source FaaS framework, specifically Google's Functions Framework. An aspect-oriented approach is adopted to transparently specify the relevant objects that should be cached, effectively decoupling function implementation from deployment configuration. The study extensively investigates various caching mechanisms, encompassing in-process, out-of-process, and network caching, and systematically assesses their impact on response times and resource utilization. The findings underscore the trade-offs inherent in employing caching techniques, ultimately aiming to optimize FaaS performance and improve overall system efficiency.**

Keywords: Cloud; Serverless; Function-as-a-Service; Cache architectures.

I. Introduction

Cloud computing solutions have gained increasing popularity over the years. Among these solutions, Function-as-a-Service (FaaS) provided by cloud platforms has emerged as a powerful paradigm enabling the implementation of scalable services triggered by external events. Functions are designed to be serverless and short-lived, allowing the infrastructure to deploy serverless applications reacting to events such as an HTTP request, uploading an image to a data lake, or sending a message to a message broker [1], [2], [3]. They are usually used in scenarios where streams of events from different sources are handled by one or more functions, writing the result to an external data repository. There are several advantages to deploying applications in FaaS platforms, including built-in scalability, improved development speed, and cost efficiency [4]. However these services present new research challenges. One of them is related to the data-intensive nature of modern systems which makes the FaaS approach heavily dependent on remote data storage access [5]. During the function's execution, latency increases with the values scaling depending on the size of data that is needed to process the function. These types of problems can be mitigated by applying caching systems [6] which bring data closer to the function's environment.

Different cache architectures have been researched and evaluated to mitigate some of the problems accessing data in FaaS [5], [7], [8]. However, to our knowledge, there is no model or implementation available to integrate and make an assessment of different cache architectures. In this work, we propose a model to integrate different cache architectures in open-source FaaS frameworks. Based on this model, we developed a configurable middleware to automatically intercept calls to FaaS functions to fetch data from the cache system. We also propose a way for system architects to choose different cache solutions during the setup phase of the middleware.

For development, we utilized the Functions Framework, an open-source framework from Google Cloud Platform (GCP). This framework is specifically designed to be compatible with GCP's Function as a Service (FaaS) platform, allowing us to demonstrate how this approach can seamlessly integrate into a production environment. The Functions Framework provides the capability to deploy a server where functions can be registered and triggered to execute custom code written by programmers. We considered three options for organizing the cache system: (i) implementing a cache within the process running the function, (ii) utilizing a cache running outside the function's process but on the same machine, and (iii) employing a cache running on a separate machine connected to the one running the function.

The system configuration remains transparent to developers as they only need to modify the configuration file and have their function return the value that should be cached during the request processing. The function does not require any knowledge about the caching mechanism as the infrastructure handles the retrieval and storage of key-value pairs for each request's content. By adhering to these requirements, the infrastructure caches the values, ensuring transparency and enhancing the usability of the developed project.

To evaluate our solution, we deployed a function that generates a thumbnail from an image [9]. The function is executed only if the thumbnail is not already in the cache. We conducted a series of experiments using various images to measure the advantages of each caching architecture and demonstrate the feasibility of implementing our proposed model. These experiments measure: (1) the time taken for the function to execute, without cache and with the three caching deployments; (2) the scalability of the system with a local and remote cache.

The paper is structured as follows: Section II provides an overview of related work, comparing it with our approach. Section III presents the architecture of our caching system and explains how it integrates with the Functions Framework. In Section IV, we provide implementation details, and Section V discusses the results obtained. Finally, Section VI presents the conclusion, along with directions for future research.

II. Related work

Serverless computing, particularly within Function-as-a-Service (FaaS) infrastructures, diverges from the conventional cloud deployment model centered around virtual machines [4], [10], [11]. This paradigm shift makes use of auto-scaling mechanisms and an architectural style based on pay-as-you-go principles. However, inherent limitations exist in this approach, with notable issues including I/O bottlenecks and the unavoidable inter-function communication occurring through slower storage channels [12], [13]. The recurrent utilization of external data sources, a common style in modern applications, introduces performance challenges by impeding client requests due to the stateless nature of functions.

Caching techniques can can be applied in order to improve the performance of the function execution, in a similar way as it is done in many other areas of computer systems that aren't directly related to FaaS, e.g. network caching services [14], IoT cloud-based solutions [15] and Content Delivery Networks [16].

The current serverless platforms use stateless containers that are ephemeral, separating functions and preventing them from sharing memory directly. This forces the user to replicate and serialize data multiple times, thereby incurring in additional performance costs. To overcome this, Shillaker and Pietzuch [17] propose Function as a Service with Memory (FaasM) , a new lightweight isolation strategy allowing functions to share memory directly and reduce resource overheads. By utilizing an abstraction similar to a distributed shared memory, FaasM makes it easier to transfer data among function instances within a worker node and across worker nodes, with the assistance of shared memory.

Mvondo et al. [8] proposed a cache system, called OFC, that uses the over-provisioned memory to reduce latency without changes at the code level. Their approach capitalizes on two prevalent characteristics in Function-as-a-Service (FaaS): the common practice of memory overprovisioning, stemming from the challenge of accurately estimating the function's effective footprint due to its input-dependent nature, and the retention of the function sandbox post-invocation to minimize cold starts. OFC is an in-memory caching system for FaaS platforms supported on RAMCloud [18] to have a distributed memory system over the worker nodes of the cluster. The prototype was developed with the Apache OpenWhisk stack [19]. Leveraging machine learning models trained on diverse input data categories, such as multimedia formats, OFC estimates the actual memory resources required for each function invocation. The surplus capacity is then allocated to the cache, optimizing overall performance.

Klimovic et al. proposed a distributed storage system named Pocket ([20]) designed for application within the domains of data analytics [21]. It operates as a distributed data store tailored to facilitate efficient data sharing in serverless analytics, with characteristics such as elevated throughput, minimized latency, and automatic scalability of computational resources. It has a storage mechanism that ensures access with sub-millisecond latency, and library with a simple API to access this storage. The system is designed to be administered by cloud service providers and adheres to a pay-as-you-utilize financial model.

Romero et al. proposed Faa$T [5], a serverless in-memory caching layer for Function-as-a-Service (FaaS) applications, that manages data accessed by the application. It eliminates the need for remote in-memory caching, reducing costs for users and FaaS providers, and enables different cache replacement and persistence policies per application. Faa$T also can remove cache space required by rarely-invoked applications from memory when not needed, reduces overall traffic to remote storage, and can pre-fetch popular data when reloading each application.

The Cloudburst [7] platform is an autoscaling Function-as-a-Service system with abstractions that enable stateful programs. It preserves the serverless computing advantages of autoscaling while offering familiar Python programming with low-latency changeable state and communication. Cloudburst uses a combination of an autoscaling key-value store and mutable caches co-located with function executors to achieve what they call logical disaggregation and physical colocation. It is built on top of Anna [22], an autoscaling key-value store for state sharing and overlay routing, and presents novel protocols that ensure consistency guarantees across function invocations that run on separate nodes.

In a related work to this topic, Wang et al. takes advantage of serverless functions itself to implement a cache system called InfiniCache [23], aiming to achieves both cost effectiveness (available in system like S3) and high-performance (available in system like Redis). They do so using techniques to provide fault tolerance to data loss, because lifetime of functions is limited, and aggregate network bandwidth of multiple cloud functions in parallel. However, applications need to be changed to use this specific cache system.

In our work we developed a generic model to integrate different caching systems for immutable data in order to reduce latency during FaaS function execution. When compared with related work, our approach is transparent to the application, without the need to change the calling function. We introduce a middleware layer between the function and the specific cache, making it possible to change the type of caching system on the setup phase.

III. Proposed Architecture

This subsection describes the generic FaaS architecture with and without caching services. Figure 1 depicts the architecture of a typical FaaS framework with its three main modules [24], [25], i.e., the API Gateway, an Event Mapper and the Function Mapper that communicate with a scheduler. The API Gateway is the endpoint to the user call that attends API requests. The Event Mapper, maps the type of event to a function. Finally, the function mapper makes the mapping to the runtime environment, effectively starting a function instance or reusing one already running. A request is received by the API and the parameters are passed to the runtime, where they are processed by a second layer of routing, named interceptor in Figure 1 which handles the unmarshalling of parameters and

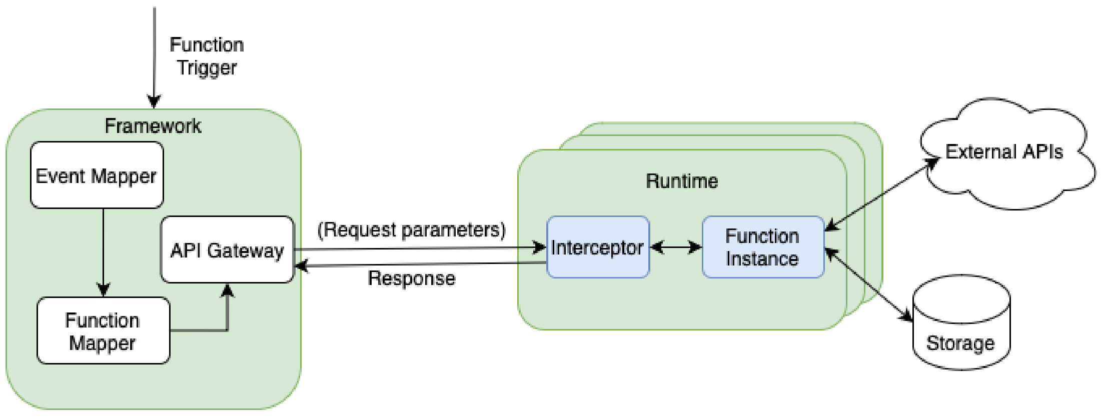


Fig. 1: FaaS architecture - without cache

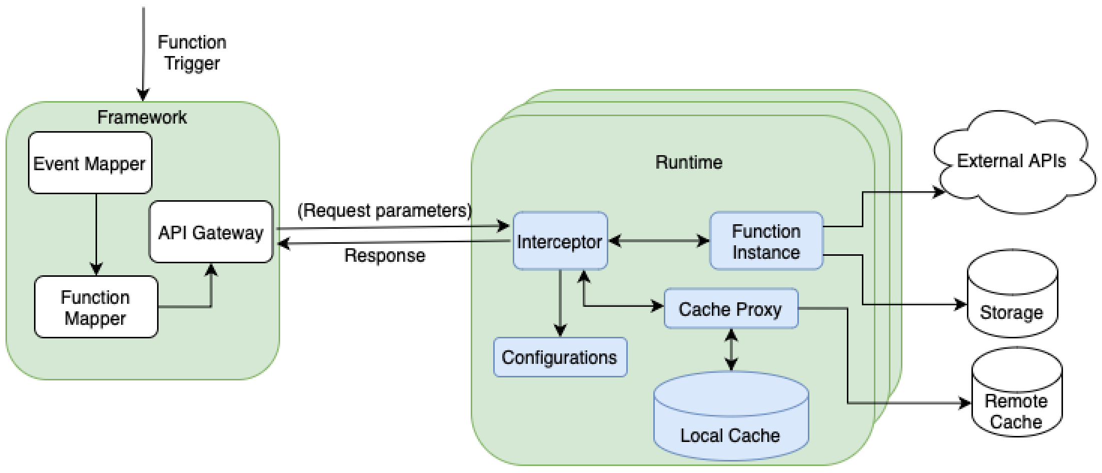


Fig. 2: FaaS architecture - with cache

calls the function code. The function has access to other APIs or external storage systems, either located in the same cloud provider or not.

Figure 2 introduces some cache-related elements near the function code, leaving the left side of the image with the basic components described previously. The proposed extensions to the baseline architecture have two levels of cache: local and remote. The local cache, which we call *in-process*, will be located in the same process as the function (and its runtime). This will allow for a very fast access to cached data. On the downside, only one function will be able to access this cached data. In a scenario with multiple functions accessing shared data, there is a remote cache, located either on another process of the same host, which we call *out-of-process*, or available by a network call, which we call *network*. This will hinder latency in the access to cached data but more functions will benefit from the presence of the shared object.

The interceptor block is responsible for the connection to the caching systems. It connects to the local or remote cache based on the type of configurations provided by the Configurations Module. Functions receive arguments and return values. Based in a declarative configuration, the interceptor determines the possible cached argument and tries to retrieve it by looking up the argument in the cache. In the function returns a new value for the object, the cache will be updated. A cache proxy is used to abstract the location, wither local or remote, and provide a uniform interface based on a key/value pair.

IV. Implementation

With the goal of creating a prototype of the system, we augmented the Google Cloud Functions Framework [26], an openly available framework. This framework operates on the principles of serverless computing and is designed to streamline the process of developing and deploying Google Cloud Functions. By using this framework, developers can compose and evaluate their functions in a local environment prior to their deployment onto the Google Cloud Platform. Consequently, this framework enables developers to harness the capabilities of widely-used programming languages such as Node.js, Java, and Go, facilitating the creation of serverless functions. Notably, the Google Cloud Functions Framework offers a user-friendly interface, managing HTTP requests and triggers, simplifying the construction and deployment of scalable cloud-centric applications. This section discusses the components that were extended in Google Functions Framework, including the dependencies of the project, composing modules, and the three caching types implemented.

A. *Extending FaaS GCP*

We have extended the FaaS GCP framework for node.js functions, incorporating a generic and dynamic caching model that is consulted before the execution of the function with the goal of reducing response times, by returning already processed values. This new middleware works with different types of cache that can be chosen by configuration. With the goal of demonstrating the benefits of our approach, we use a common use case based on a thumbnail generation scenario, using an HTTP trigger, which is a based on the referenced use cases proposed by Yussupov et al. [9].

One of the concerns in the development of this solution is the transparency for the programmer [27]. In this case, the image content is an argument of the function to be executed, being in the request object. However, the property may have any name, so a JSON file is used to declare which property name is relevant in the context of the request. This approach is based on techniques used in aspect-oriented programming [28], where behavior is added to existing code, without explicit code modification. Doing so, makes it possible to have the configuration deployed along with the code, but keeps the details of the source code independent from this aspect. When the function returns an object that should be used to update the cache, the extended framework takes care of setting the key/value pair, following the commitment of maintaining the framework transparent and independent of the function's code.

The properties of the configuration file are presented in Listing 1. The property value "image" is used to define the property name that is used in the request to represent the image content. The `cachingService` property from the configuration file is used by the *cache proxy* to define the type of cache that is being used, either the in-process cache or the remote cache, based on the Redis cache [29]. If it is the Redis remote caching system, the `redisRemoteConnectionString` property is used to define the connection string to the caching service.

```
{
    "property": "image",
    "cachingService": "nodeLocal",
    "redisRemoteConnectionString": "redis://****"
}
```

Listing 1: Configuration File

GCP Functions Framework together with node-cache and Redis modules accomplished the thumbnail generation use case where an image is uploaded, and if the image is already in the cache, the thumbnail of that image is returned, if not, the function's execution is triggered and the thumbnail has to be generated and cached. In addition to the base code of the framework we used the following packages:

1) Crypto [30], a module used to generate the hash keys for the cache Key-Value pairs that are stored;
2) Image-Thumbnail [31] the module that enables the generation of the *fingerprint* given the original image, used on the testing and evaluation of the solution;
3) Node-Cache [32], is an in-memory caching system used to store key-value pairs, used as the in-process cache implementation;
4) Redis [33], [29] is an open-source in-memory data structure that can be used as a caching service, message broker, streaming engine, or database.

B. *Cache Proxy*

The decision of using an *in-process*, *out-of-Process*, or *network* caching service occurs on the function wrapper module where the implementation of the Interceptor is made. Operation `ConnectCache`, used by the Interceptor module, connects to a particular cache implementation. As seen in Listing 2, the caching options are `nodeLocal`, `redisLocal` or `redisRemote`, where the default case uses the connection to the `redisLocal`. The connection on `ConnectCache` operation only occurs if the `req.cachingClient` property is null, which means that a connection to the caching system wasn't yet defined, and a connection should be established. This way the same connection is reused, saving limited resources.

```
async function ConnectCache(req, cachingService) {
  if (req.cachingClient == null) {
    switch (cachingService) {
      case "nodeLocal":
        cacheProvider.start()
          req.cachingClient = cacheProvider
          break;
        case "redisLocal":
          var redisClient = redis.createClient({});
          await redisClient.connect()
          req.cachingClient = redisClient
        case "redisRemote":
          var redisClient = redis.createClient({
            url: propertyAccessTest.
                redisRemoteConnectionString
          })
          await redisClient.connect()
          req.cachingClient = redisClient
          break;
        default:
          var redisClient = redis.createClient({});
          await redisClient.connect()
          req.cachingClient = redisClient
    }
    return req.cachingClient
  }
}
```

Listing 2: Cache connection details

C. *Functions Manager*

Listing 3 contains the summary of an extension to the framework function wrapper, which handles the HTTP request. When a request arrives with the relevant property on the event body, a cache key is generated using an SHA256 hash function [34] for the base64 value of the image, generating a unique key per image. The key, the cache proxy and the request object are then passed to the Functions Manager module. On the Functions Manager, the cache proxy is used to access and check if the key value is already cached. If the value is cached, then the cache proxy accesses the caching system and returns the value, which is then returned to the client that originated the request. Otherwise, if the key isn't found on the cache, the function is executed. On our use case, the function generates the image *thumbnail* and the cache is written using by the cache proxy instance. Finally, the thumbnail value is returned to the client.

```
const wrapHttpFunction = (func_code): reqHandler =>
{
  return (req: Request, res: Response) => {
    // collect data from request
    image = getImageFromBody(req.body[configfile.
        property])
    // create hash key
    hash_key = crypto.createHash('sha256')
      .update(image)
      .digest('hex');
    // connect to cache
    req.cachingClient = await cache_proxy
      .ConnectCache(req, configfile.cachingService)
    // get value from cache
    functionResult = await cache_proxy.GetValueByKey
        (configfile.cachingService, hash_key)
    if (functionResult == undefined) {
      functionResult = await func_code(req, res);
      cache_proxy.SetValue(configfile.cachingService
          , hash_key, functionResult)
    }
    // returned cached or live value
    return functionResult
  }
};
```

Listing 3: Function Wrappers

a) In-Process Implementation: The *in-process* cache depends on the node-cache module [32], the module allows the creation of a caching service instance that is executed on the same process where the function instance will also be executed. The node-cache used as the *in-process* data storage is an in-memory caching service where key-value can be stored. The module supports the association of a string to a JSON object and the definition of an expiration time best known as time to live. The interceptor calls the functions manager module, passing the cache proxy as a parameter. The manager module then invokes the `GetValueByKey` method, using the pre-calculated hash value of the image. This allows it to access the specific cache and check whether the value is already cached for retrieval or if the function code needs to be calles.

b) Out-of-Process Implementation: The *out-of-process* case has the caching service system running on the same machine as the function is being executed, so that multiple instances of the function can access the cache on that machine. This scenario was designed to support vertical scalability, where more resources are allocated to the same machine since the caching service isn't shared along different machines.Redis [29] is the caching service used in the testing scenarios. The data structure is composed by key-value pairs that are stored on an in-memory dataset, which enables low latency and high throughput data access. In this implementation, the only main difference is on the cache proxy, where the connection occurs to the Redis cache system and not the node-cache instance that is running on the same process as the function.

c) Network Implementation: For the *network* case, the caching service system runs on the server located in the same cluster where are the computational nodes supporting the execution of the function. This means the cache service can be accessed by different nodes running the one or more functions, and accessing the same cache, with values that could have been processed by requests to other clients. This scenario gives the most advantages for situations where scalability is necessary to guarantee a good quality of service, at the cost of some aditional latency. The next steps are the evaluation of results for the proposed scenarios with a brief description of the setup that is discussed in the next section.

V. Evaluation

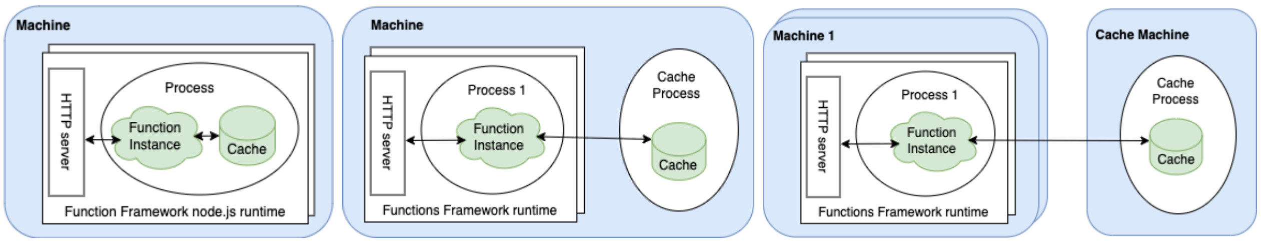
The development of the three previously described cache policies and deployments introduces potential performance advantages that need to be assessed. This comparison between the *in-Process*, *out-of-Process*, and *network* will give an understanding of how the latency, depending on the proximity of the Functions Framework instance to the caching system, impacts on the function execution time. Furthermore, this section discuss how the a cache deployment, either local or remote, may influence the scalability of the server where the function code is deployed.

A. Evaluation Setup

The *in-process* use case is considered the simplest one, where the image processing function and the caching service are running on the same process as seen in Figure 3a. This case has the lowest latency values in cache access from the three different cases. The *out-of-Process* use case has two different processes running, the first one accommodates the function that processes the client's requests, and the second one that has a Redis [33] instance running, working as the caching system with the key/value store of the hashed image and the thumbnail value. For this case, since the processes are different, it is expected that the latency for the cache access process to have a bigger value than the *in-process* use case. Figure 3b depitcs the interaction between the function instance and the cache, where each one of them is executing on their respective process, inside the same computer node.

Regarding the *network* use case, there are two computation nodes, one where our function is being executed and attends the HTTP requests, consulting the second virtual machine where there is a Redis server instance with the cached data as represented on Figure 3c. There could be multiple clients accessing the virtual machine that has the Redis cache instance, taking advantage of the caching service, where images that are frequently accessed by different clients could have already been processed, obtaining the result faster by accessing the cached thumbnail value.

Tests were made using three different images sizes, in order to better understand the impacts it creates on the functions framework system. The image sizes are 18KB (small), 200KB (medium) and 4MB (large). Tests were done with two types of virtual machine deployed at the Google Cloud Platform. The one running the Functions Framework runtime and the thumbnail use case function is a type `e2-micro` , a machine with 0.25 vCPU and a total of 1 GB memory. The other type of virtual machine is a `e2-medium`, with 0.5 vCPU, and 2 GB memory. In the VM, a docker container is running the Redis server supporting the *network* cache.



(a) In-Process cache deployment (b) Out-Of-Process use case diagram (c) Network use case diagram

Fig. 3: The three evaluated deployment strategies

B. Cache impact on performance

The first test evaluates the latency introduced by the caching system, assessing the function execution performance. The values are retrieved between the moment where it is validated if the image hash key is already cached and the moment after the thumbnail is generated or the cached value is retrieved and returned.

Table I refers to small image, with 18KB. The table is composed of 5 columns. The first one states the image size in bytes, and the second one is the execution time when the image thumbnail isn't cached. As for the third, fourth, and fifth columns, they represent the time between evaluating the key and returning the already cached thumbnail value respectively for each of the cases.

The in-process cache access latency is the smallest one, having the out-of-process scenario a cache access time 4x bigger and the Network case 150x compared to the in-process case. However, in all three cases, using a cache represents an advantage when compared to the computation of the thumbnail of the image. The results show that the closer the cache is to function, the smaller will be the latency values. The Network case has the biggest discrepancy on this test since the latency isn't zero in the communications across the network between different GCP virtual machines.

This second round of tests resorts to a bigger image, nearly 13 times bigger (228902 bytes image) than the first. Table II summarizes the results for this image size. Compared to the results in Table I case, the image processing time increased while the Time to access the caching services were nearly the same.

As for the large image, the processing time in Table III, are higher given the extra memory and CPU resources needed per request. Looking at the worst latency scenario, the network cache, there will be an improvement in more than one order of magnitude when compared with the no cache scenario.

C. Cache impact on scalability

A local or remote cache can have an impact on the scalability of the server running the Functions Framework runtime. To measure this impact we used a client-side tool, Apache JMeter [35], to make several requests to the function deployed at the extended Functions Framework runtime running on a virtual machine. This experiment goal was to observe the *saturation point* of the server, that is, the point where CPU and memory are exhausted and requests are being throttled by the infrastructure. Figure 4 represent an in-process cache deployment, while Figure 5 represents the network cache deployment. In both cases the client-side tool makes a total of 750 HTTP requests, with 5 threads on the client-side to explore the concurrent handling of requests on the server-side. Each HTTP request transports the medium size image in the request body.

Figure 4 shows the server can handle up to 15 requests per second for 30 seconds, after which the resources become congested and drops the processing rate to approximately 5 hits per second. This is a consequence of having the runtime, function code and cache on the same process. The response time is the fastest, as seen in Section V-B, but the server overflows quickly.

Figure 5 shows that the server can sustain a number of hits per second between 12 and 15 until the end of the requests. This is possible because the cache is located outside of the process, in a remote machine, and is the trade-off of spending more time accessing the remote object, as discussed in Section V-B. However, in cases of higher loads, such as this, the remote cache allows the function to continue serving requests.

VI. Conclusion

Challenges associated with efficient data caching were analyzed and the solution Efficient FaaS was presented, an adaptation of the Functions Framework from the Google Cloud Platform for serverless jobs execution. Efficient FaaS aims to provide a simple way to integrate different cache systems, in this work we integrated three types of caching systems that have their pros and cons.

The evaluation shows that the solution has high performance for images with lower sizes and is able to process a high number of requests on machines with low capacity. The network use-case shows that is possible to have an external cache and several functions instances running in different machines. This is important for scenarios where the cloud provider can change the number of machines supporting the function execution in a elastic manner, although the current evaluation does not address this issue.

As for future work, we intent to test different types of cache systems and caching policies. The caching policy disposal [36] should be addressed, because the cache can't grow indefinitely given memory limitations inherent in every system [37], [38], [39]. This issue can be solved by giving an expiration parameter to the cached values, based on the frequency that a client needs to access those keys [40].

TABLE I: 18 KB image processing times

	Size (Bytes)	**No Cache (ms)**	**In-Process (ms)**	**Out-of-Process Redis same VM (ms)**	**Network (ms)**
	18617	39.112	0.159	0.709	27.865
	18617	36.359	0.157	0.904	31.121
	18617	36.755	0.210	0.541	27.772
	18617	40.679	0.172	0.521	27.791
	18617	36.637	0.370	1.063	26.825
	18617	36.837	0.156	0.708	24.244
	18617	35.661	0.117	0.543	24.013
	18617	36.917	0.122	0.853	30.477
	18617	36.608	0.143	0.483	29.600
	18617	39.776	0.217	0.473	23.943
Average		36.796	0.158	0.626	27.781
Standard Deviation		0.332	0.029	0.118	1.961

TABLE II: 200 KB image processing times

	Size (Bytes)	**No Cache (ms)**	**In-Process (ms)**	**Out-of-Process Redis same VM (ms)**	**Network (ms)**
	228902	50.544	0.162	0.646	25.512
	228902	45.138	0.158	0.780	25.639
	228902	46.836	0.169	0.576	25.915
	228902	46.820	0.148	0.644	25.075
	228902	45.812	0.154	2.808	25.164
	228902	46.191	0.166	0.815	25.109
	228902	44.815	0.306	0.650	24.619
	228902	46.296	0.115	0.629	25.371
	228902	49.345	0.155	0.774	25.392
	228902	46.512	0.287	0.531	24.378
Average		46.404	0.160	0.648	25.267
Standard Deviation		2.016	0.062	0.057	0.567

TABLE III: 4 MB image processing times

	Size (Bytes)	**No Cache (ms)**	**In-Process (ms)**	**Out-of-Process Redis same VM (ms)**	**Network (ms)**
	4106938	352.747	0.231	1.452	24.722
	4106938	354.811	0.209	1.310	25.756
	4106938	328.403	0.217	2.723	25.820
	4106938	349.864	0.246	1.771	25.190
	4106938	433.565	0.400	1.999	24.935
	4106938	362.136	0.226	7.864	25.359
	4106938	349.615	0.381	19.008	24.359
	4106938	319.041	0.202	1.617	25.268
	4106938	316.708	0.204	1.446	25.126
	4106938	372.713	0.192	1.520	24.445
Average		351.306	0.222	1.694	25.158
Standard Deviation		9.983	0.019	0.034	0.139

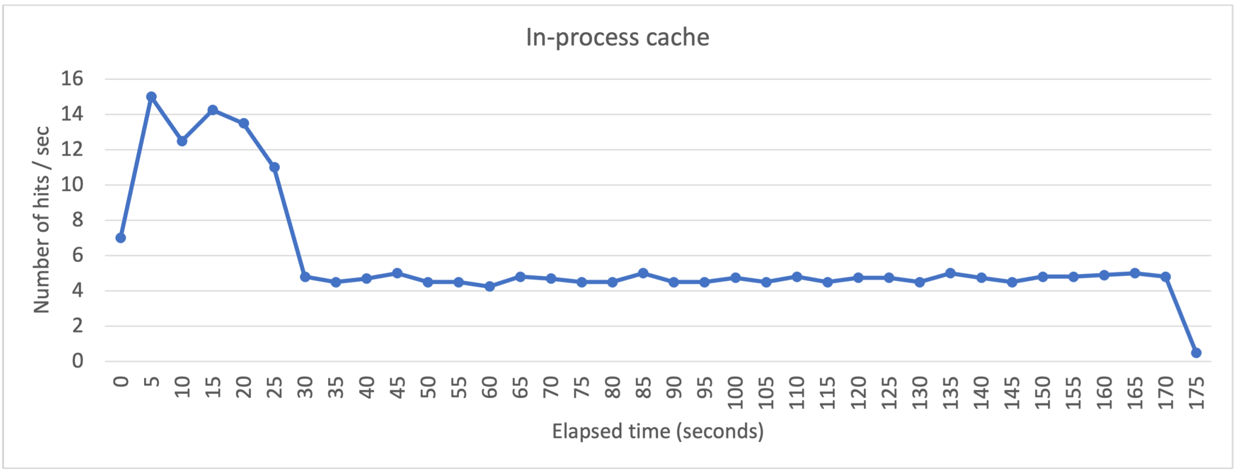


Fig. 4: Medium image processing with in-process cache

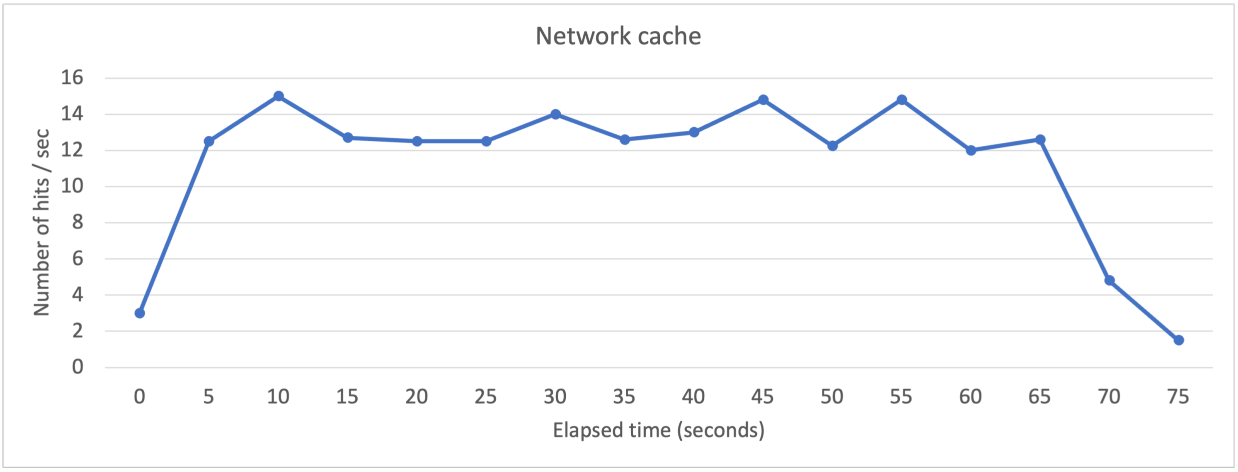


Fig. 5: Medium image processing with network cache

References

[1] A. Tzenetopoulos, E. Apostolakis, A. Tzomaka, C. Papakostopoulos, K. Stavrakakis, M. Katsaragakis, I. Oroutzoglou, D. Masouros, S. Xydis, and D. Soudris, "Faas and curious: Performance implications of serverless functions on edge computing platforms," in *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt Am Main, Germany, June 24 – July 2, 2021, Revised Selected Papers*, (Berlin, Heidelberg), p. 428–438, Springer-Verlag, 2021.

[2] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*, pp. 1–20. Singapore: Springer Singapore, 2017.

[3] P. Rodrigues, F. Freitas, and J. Simão, "Quickfaas: Providing portability and interoperability between faas platforms," *Future Internet*, vol. 14, no. 12, 2022.

[4] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[5] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, (New York, NY, USA), p. 122–137, Association for Computing Machinery, 2021.

[6] M. M. Rovnyagin, V. K. Kozlov, R. A. Mitenkov, A. D. Gukov, and A. A. Yakovlev, "Caching and storage optimizations for big data streaming systems," in *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pp. 468–471, 2020.

[7] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, p. 2438–2452, jul 2020.

[8] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "Ofc: An opportunistic caching system for faas platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, (New York, NY, USA), p. 228–244, Association for Computing Machinery, 2021.

[9] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*, pp. 273–283, ACM, Dec. 2019.

[10] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, (USA), USENIX Association, 2020.

[11] J. Simão, J. Singer, and L. Veiga, "A comparative look at adaptive memory management in virtual machines," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, pp. 452–457, 2013.

[12] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 181–188, 2018.

[13] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *CoRR*, vol. abs/1812.03651, 2018.

[14] M. Kataoka, K. Toumura, H. Okita, J. Yamamoto, and T. Suzuki, "Distributed cache system for large-scale networks," in *2006 International Multi-Conference on Computing in the Global Information Technology - (ICCGI'06)*, pp. 40–40, 2006.

[15] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *2020 IEEE International Conference on Fog Computing (ICFC)*, pp. 17–24, 2020.

[16] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Mind the cache: Large-scale explorative study of web caching," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, (New York, NY, USA), p. 2497–2506, Association for Computing Machinery, 2019.

[17] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433, USENIX Association, July 2020.

[18] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, aug 2015.

[19] "Apache openwhisk.." `https://openwhisk.apache.org/`. Accessed: 2023-05-12.

[20] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless

analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[21] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (USA), p. 193–206, USENIX Association, 2019.

[22] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 401–412, 2018.

[23] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 267–281, USENIX Association, Feb. 2020.

[24] J. Wirth, "Anatomy of a functions as a service (faas) solution," Apr 2020.

[25] E. van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, and A. Iosup, "The spec-rg reference architecture for faas: From microservices and containers to serverless platforms," *IEEE Internet Computing*, vol. 23, no. 6, pp. 7–18, 2019.

[26] Google Cloud, "Functions framework documentation." `https://cloud.google.com/functions/docs/functions-framework`. Accessed: 2023-07-23.

[27] S. M. Sadjadi, P. K. McKinley, and B. H. C. Cheng, "Transparent shaping of existing software to support pervasive and autonomic computing," in *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, DEAS '05, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2005.

[28] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," *ACM Trans. Program. Lang. Syst.*, vol. 26, p. 890–910, sep 2004.

[29] Redis, "Redis Documentation." `https://redis.io/`. Accessed on: 2023-08-01.

[30] "Node.js Crypto API." `https://nodejs.org/api/crypto.html`. Accessed on: 2023-07-17.

[31] "image-thumbnail - npm package." `https://www.npmjs.com/package/image-thumbnail`. Accessed on: 2023-07-17.

[32] node-cache, "node-cache GitHub Repository." `https://github.com/node-cache/node-cache`. Accessed on: 2023-07-17.

[33] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo, "Towards scalable and reliable in-memory storage system: A case study with redis," in *2016 IEEE Trustcom/BigDataSE/ISPA*, pp. 1660–1667, 2016.

[34] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," *Federal Information Processing Standards Publication*, vol. 180-4, 2015. Accessed on: 17 05 2023.

[35] "Apache JMeter - Apache JMeter™." Accessed: 2023-07-23.

[36] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache replacement policies revisited: the case of p2p traffic," in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pp. 182–189, 2004.

[37] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *SIGARCH Comput. Archit. News*, vol. 35, p. 381–391, jun 2007.

[38] T. Johnson, D. Connors, and W. Hwu, "Run-time adaptive cache management," in *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, vol. 7, pp. 774–775 vol.7, 1998.

[39] M. Qureshi, D. Thompson, and Y. Patt, "The v-way cache: demand-based associativity via global replacement," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 544–555, 2005.

[40] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," *SIGMETRICS Perform. Eval. Rev.*, vol. 18, p. 134–142, apr 1990.