INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Department of Electronics and Telecommunications and Computer Engineering

Reducing Execution Time in FaaS Cloud Platforms

Ivan André Gomes Rosa

Bachelor of Engineering

Dissertion to obtain the Master's degree
in Informatics and Computers Engineering

Orientadores : Professor Doutor Filipe Freitas
Professor Doutor José Simão

Júri

Presidente: Professor Doutor Nuno Cruz

Vogais: Professor Doutor Manuel Barata
Professor Doutor José Simão

December, 2022

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Department of Electronics and Telecommunications and Computer Engineering

Reducing Execution Time in FaaS Cloud Platforms

Ivan André Gomes Rosa

Bachelor of Engineering

Dissertion to obtain the Master's degree
in Informatics and Computers Engineering

Orientadores : Professor Doutor Filipe Freitas
Professor Doutor José Simão

Júri

Presidente: Professor Doutor Nuno Cruz

Vogais: Professor Doutor Manuel Barata
Professor Doutor José Simão

December, 2022

Abstract

The increase in popularity of code processing and execution in the Cloud led to the awakening of interest in Google's Functions Framework, with the main objective being to identify possible improvement points in the platform and its adaptation in order to respond to the identified need, also obtaining and analysing the results in order to validate the progress made.

As a need for the Google Cloud Platform Functions Framework, it was found that an adaptation would be possible in order to promote the use of cache services, thus making it possible to take advantage of previous processing of the functions to accelerate the response to future requests. In this way, 3 different caching mechanisms were implemented, In-Process, Out-of-Process and Network, each responding to different needs and bringing different advantages.

For the extraction and analysis of results, Apache JMeter was used, which is an open source application for implementing load tests and performance measures of the developed system. The test involves executing a function to generate thumbnails from an image, with the function running in the framework. For this case, one of the metrics defined and analyzed will be the number of requests served per second until reaching the saturation point.

Finally, and based on the results, it was possible to verify a significant improvement in the response times to requests using caching mechanisms. For the case study, it was also possible to understand the differences in the processing of images with small, medium and large dimensions in the order of Kbs to a few Mbs.

Keywords: Cloud; Functions Framework; Google; Processing; Cache; Load; Performance; Image; Function.

Resumo

O aumento de popularidade do processamento e execução de código na Cloud levou ao despertar do interesse pela Functions Framework da Google, sendo o objetivo principal identificar pontos de possível melhoria na plataforma e a sua adaptação de forma a responder à necessidade identificada, tal como a obtenção e análise de resultados com o objetivo de validar a progressão realizada.

Como necessidade da Functions Framework da Google Cloud Platform verificou-se que seria possível uma adaptação de forma a promover a utilização de serviços de cache, possibilitando assim o aproveitamente de processamentos prévios das funções para acelerar a resposta a pedidos futuros. Desta forma, foram implementados 3 mecanismos de caching distintos, In-Process, Out-of-Process e Network, respondendo cada um deles a diferentes necessidades e trazendo vantagens distintas entre si.

Para a extração e análise de resultados foi utilizado o Apache JMeter, sendo esta uma aplicação open source para a realização de testes de carga e medidas de performance do sistema desenvolvido. O teste envolve a execução de uma função de geração de thumbnails a partir de uma imagem, estando a função em execução na framework. Para este caso uma das métricas definidas e analisadas será o número de pedidos atendidos por segundo até atingir o ponto de saturação.

Finalmente, e a partir dos resultados foi possível verificar uma melhoria significativa dos tempos de resposta aos pedidos recorrendo aos mecanismos de caching. Para o caso de estudo, foi também possível compreender as diferenças no processamento de imagens com dimensão pequena, média e grande na ordem dos Kbs aos poucos Mbs.

Palavras-chave: Cloud; Functions Framework; Google; Processamento; Cache; Carga; Performance; Imagem; Função.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface. 5, 8

CLI Command-Line Interface. 7, 37
CPU Central Processing Unit. 51

FaaS Function-as-a-Service. 1, 2, 5, 6, 7, 9, 10, 11, 12, 13, 15, 16, 35, 57

GCP Google Cloud Platform. 2, 3, 7, 8, 12, 38

HTTP Hypertext Transfer Protocol. 1

JSON Javascript Notation Object. 8

VCPU Virtual Central Processing Unit. 36
VM Virtual Machine. 17
VMS Virtual Machines. 9

1

Introduction

In this chapter it is explained some of the Cloud main advantages, and it evolves into the motivation that brought the idea of developing this project. It aims to summarize the proposed solution and the contributions that the development brings to the project, and finally focusing on the document organization.

1.1 Work Context

The Cloud computing solutions have turned increasingly more popular over the years for it's ease of use, scalability properties and disassociation between infrastructure management and software development. The FaaS is a type of service that cloud providers make available to implement scalable services that are executed based on external triggers [1], for example an HTTP request, blob file upload along some others. "From the perspective of a cloud provider, serverless computing provides an additional opportunity to control the entire development stack, reduce operational costs by efficient optimization and management of cloud resources (...) and manage cloud-scale applications" [55]. Accordingly to Wang Ao et. all (2021), "Function-as-a-Service (FaaS), enables a new way of building and scaling applications" [54], that are considered adequate to answer problems that require scalable resources, being the typical example, images, video processing and large scale transaction systems.

As for the FaaS, it is a service available on cloud providers platforms that facilitates the execution and development of application functionalities "where software engineers can focus on business logic and leave the infrastructure management" [60] complexity where these solutions are executed to cloud providers. The use cases can be associated with an on-demand execution, where the service is off until any type of trigger is received, that starts the execution of the FaaS code with the received data parameters.

1.2 Motivation

Although there are several advantages from deploying application in FaaS platforms, including built-in scalability, improved developer speed, since there is no need to worry about the server or deploys and cost efficiency [61], these services have some disadvantages. One of them is the need for remote data storage access. These remote accesses are essential to the operation of a typical FaaS function given the R nature of them [53]. During function's execution, latency increases with the values scaling depending on the size of data that is needed to process the function. These type of problems can be mitigated by applying caching systems [39] which bring data closer to the function's environment.

As FaaS solutions are becoming increasingly popular, different cache architecture have been researched and evaluated to mitigate some of the problems accessing data [52]. However, to our knowledge, there is no model or implementation available to integrate and make an assessment of different cache architectures.

1.3 Proposed Solution

In this work, we propose a model to integrate and evaluate different cache architectures in open-source FaaS frameworks [42]. Based on this model, we developed a configurable middleware [34] to automatically intercept calls to FaaS functions in order 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.

As for the development, we used an Open Source Framework from Google Cloud Platform (GCP), Functions Framework. This framework gives the possibility of deploying a server where a function can be registered and triggered to execute any code that the programmers writes.

Three options were considered to organize the cache system: (i) a cache inside the process running the function; (ii) a cache running outside the process of the function but in the same machine; (iii) a cache running on a different machine connected to the one running the function.

To evaluate the middleware, a use case was developed, consisting on generating thumbnails from images [77], being the function triggered only if the image's *fingerprint* isn't already in cache. A set of experiments were made to measure the benefits of each cache architecture.

All of the data stored on the caching system is immutable [45] since for the same request an equal response should be sent to the client.

The configuration of the system is transparent to the developer, since he only needs to change the configuration file, and on his function to return the value that should be cached on the request attendance. The function doesn't need to have information about the cache, since the infrastructure does the job of getting and setting the key and value for the content of each request. By following these requirements, the values are cached on the infrastructure side, giving the transparency needed to ease the usability of the developed project.

1.4 Contributions

The main contributions of this work are:

1. A generic model that shows how to integrate different cache architectures in FaaS Frameworks.
2. A configurable middleware to work in GCP Functions Framework that allows systems architects to choose a cache system at the setup phase.
3. The implementation and evaluation of three cache architectures using GCP Functions Framework.

1.5 Document structure

The current document is organized in 3 different main chapters, representing a typical structure of Introduction, Related Work and finally an introduction to what is being development and more details about the already developed use case.

First chapter contains an Introduction for the theme of the thesis and the challenge to accomplish better results with the proposed developments.

Second chapter resumes the definition and use cases of Function as a Service, also providing a brief description of both studied frameworks.

The third chapter has information about published papers with work related to the problem. It starts with the introduction of the framework's architecture, the proposed solution and how the use case was implemented.

2

FaaS Background and Related Work

2.1 FaaS Background

FaaS follows an event driven execution model. In this type of model, developers don't have to worry about low-level details such as infrastructure management, security updates, availability, reliability or server monitoring since that is a responsibility of the cloud provider where our function is hosted. This disassociation greatly improves developers rate of code development, since any other necessity rather than coding is answered by the cloud provider.

One of the greatest advantages of the serverless model has over the serverful is the pay-as-you-go billing system, where the customer only pays for the resources used during the time that the function is executing. The time frame of execution time goes to the hundreds of millisecond intending to deliver an accurate invoice to the client.

The main use cases for this type of solutions include event-driven or flow-like processing patterns [10], API where the flow of data is controlled between different services and scientific cloud computing [16], since there are no worries about scaling on heavy processing jobs.

However it is known that by gaining in terms of ease of use, flexible scaling, fine grained pricing model and availability of service there will be trade-offs, mainly at the service quality level in terms of response times to function requests. This problem is mainly caused, because complex systems need to store data, and by decoupling the

computing layer from other layers in order to ease the scale up/down [74], the system will need to make a remote call to the storage layer in order to obtain data that is used to attend the request creating an overhead to the response time that degrades performance.

2.2 Frameworks

A framework [57] defines a software structure that enables the production of modules or artifacts, which can serve as the foundation to develop new software. It provides a standard way to design, build and deploy applications on a reusable environment. In a simpler way it defines a set of tools used to develop other software applications.

A Function as a service is an on-demand infrastructure that enables the execution of code. In this case it is possible to plan only the application, without worrying about infrastructure problems, the entity in charge of these type of questions is the Cloud Provider.

For the project two different approaches were made. In the first one, Open FaaS [37] was used in order to better understand how a simple function was deploy on the environment and how did the trigger passed through the modules that compose the framework. As for GCP Functions Framework, it was explored and adapted to accomplish the proposed objectives.

These frameworks are essential to study the possibility of implementing a caching solution in order to mitigate the latency of function's execution, caused by the access to external data storages.

2.2.1 Open FaaS

Open FaaS is an open source project [51] that simplifies the deployment of event-driven functions and microservices into a container using kubernetes [50, 63].

It can be executed in any cloud environment without the vendor lock-in problem [12]. Vendor lock-in is a barrier to the heavy adoption of cloud computing, it is caracterized by cases where it is challenging to transition to a competitor's service, regularly the transition incurring in a major adaptation of the system.

The platform supports any language for the function that has to be executed based on the trigger event, since it will be packaged in a Docker container, following the most recent trends where cloud functions are deployed on containers [68], the dependencies

needed to execute that function will be downloaded if defined in the docker file. As another advantage, it is scalable depending on the traffic spikes, also scaling down if it goes idle.

The framework creates 1 to many containers depending on the scaling parameters provided by the user. It provides an API gateway that triggers the functions based on user requests that will eventually be processed by one of the containers instantiated by Kubernetes or other hosting systems.

For Mac OS or Linux in order to deploy the container first it is needed to install the command line interface of Open FaaS. Then install helm, and start a minikube instance in docker [43] with the command 'minikube start'. After this steps proceed to create namespaces of OpenFaaS core components and functions. Add the helm repository, update the charts using helm. Generate a random password and a secret for it, then install OpenFaaS using the chart, set the Url as an environment variable and finally login using the CLI once all Kubernetes pods are started.

2.2.1.1 Architecture

The Open FaaS architecture [67] is composed by a function watchdog that allows HTTP requests to be forwarded to the target process, triggering the function execution. It also contains an API Gateway that, just as the function watchdog it processes the HTTP requests, but additionally scales functions by changing the replica count.

Monitorization of the system is based on a Prometheus/Grafana module [40] that contains information about function rate, replica scaling and execution time of the function as a service.

Finally all of these modules are supported by a docker container. The container consists on a standalone, lightweight executable package of software that runs all of the applications needed to support Open FaaS framework execution.

2.2.2 GCP Functions Framework

The GCP Functions Framework is a serverless environment that runs a function, which responds to different types of triggers.

Just like OpenFaaS, Functions Framework provides the serverless solution that works as a black box for the developer who only submits the code of the application, and it gets executed upon previously defined triggers. Normally the user only has control over limited configurations through which performance can be controlled but since the

framework is open source, it provides the possibility to open the "box" and adjust the code based on the needs for the project.

The framework supports Cloud Events and a Pub/Sub [58] emulator. Messages with JSON objects can be published on a queue that will eventually be processed and trigger any function that is defined. This mechanism gives the possibility to accommodate a higher number of requests, if the user doesn't need the response immediately.

2.2.2.1 Deployment

In order to start using the framework, it is only needed to install the npm package with the command: 'npm install google-cloud/functions-framework', which creates a node package with editable files that compose the framework. The next step is to create a function that will be triggered by an event and deploy it by running the command: 'npx google-cloud/functions-framework –target=functionName'.

As a first approach the Handler that answers the trigger was intercepted and a new property was added to the request parameter in order to receive it on the function that is triggered.

2.2.3 Framework decision

As the study of frameworks continued, it was decided that GCP Functions Framework should be used, since the approach to the problem could be made in a simpler way. It is a serverless environment that enables the connections between application with different triggers. There are many types of triggers [1] for these functions such as HTTP requests, Pub/Sub topics, google cloud storage upload. These triggers are called events that will give the order of execution to the function that is defined on the framework.

During the first contact with GCP Function Framework, the objective was to trigger the function and obtain a response to the function's request from an external API. The referenced image in Figure 2.1 represents the workflow of the GCP functions framework, from the creation of the instance to the request that is made by the function on this first simple approach.

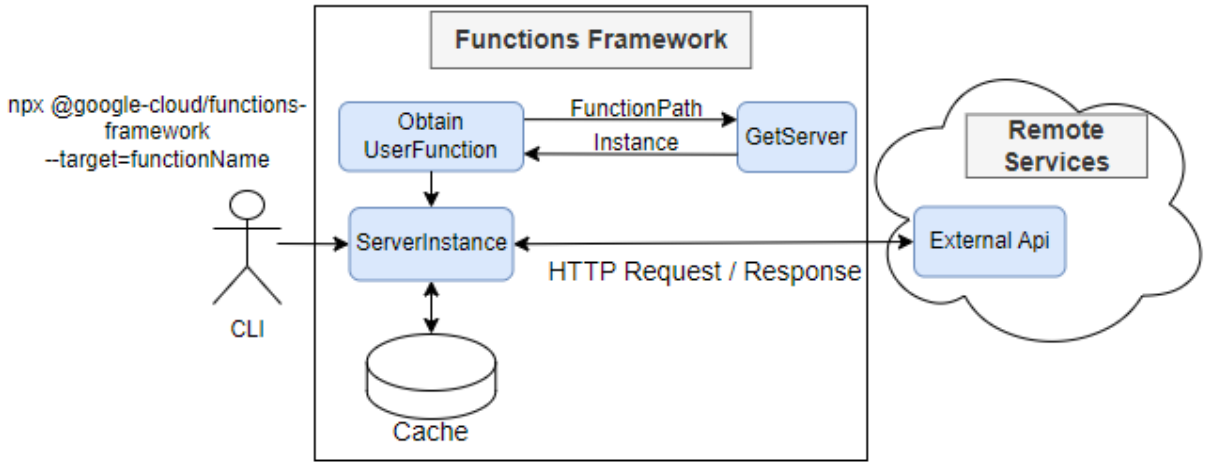

Figure 2.1: Functions Framework function execution workflow without cache

The functions framework architecture contains components with functionalities to obtain the user function passed as parameter when the framework is started. Subsequently this component gets an instance of the server that gives a response to the user request, keeping the sandbox alive to hasten the setup process [23, 29] for serving future invocations. Figure 2.1 represents a simplified version of the interaction between a function instance and an external API, with the blocks that support the instantiation of the function based on the passed parameters.

The section 3.1 gathers more detail about the GCP Functions Framework architecture, diving into the architecture and the adaptations that were made.

2.3 Problems and challenges in FaaS

With the development of projects that use FaaS as a solution, the difficulties and challenges imposed by the implemented systems were noticed and improved solutions are being studied and developed to mitigate them.

In order to keep FaaS easily maintained and scalable, some solutions that require persistent state are hard to implement, since new instances would have to obtain this state from a shared datasource, impacting the new instances deployment time for example.

- **State management** is indispensable in stateful applications [48], where a short message representing a state needs to be sent across different components. This state can be stored in the function, but this won't be replicated across other VMS

that can instantiate the original image as a response to a scalability necessity, causing the problem of state management.

- **Networking**, systems have to access shared storage to pass state between cloud functions, which impacts latency values. Users also don't decide where functions run, excluding the possibility of optimization in requests.
- **Predictable performance**, since users are denied of control over resources, there can be bad timings that cause queuing delays, causing worst response times of applications. The reassignment of resources from customer to customer are also unpredictable, this event is called "cold start" [38], and causes latency during the process of preparing the software environment of the function to start attending requests.

The caching solution is directly related to some of the topics discussed above. Every request routed to the FaaS function, having the necessity to obtain data from another system, has to make a request that causes latency, with a higher or lower degree based on different parameters. These can be: the data that has to be transferred [18], network rates, distance between the machines causing impact on the function's performance.

Caching gives the possibility to reduce "cold starts" in cases where new instances need data from external sources. Also reduces the response time of functions that would rather have to make requests to external servers, that would take much longer than respond with cached data. The solution of the problems described above allows a wider range of problems to be solved via serverless solutions.

2.4 Related work

Due to the stateless property of FaaS, these types of solutions must interact frequently with external data storage, limiting the performance of the system during client requests.

Mitigation methods can be applied by using caching services in order to improve the performance of the execution of function's that return immutable data to equal client requests. Caches are used in many different areas of study that aren't directly related to FaaS, as an example for network caching services [31], IOT cloud-based solutions [41] and medical services [47].

The main performance limitations identified in FaaS [38, 49] are the scheduling latency, which garnered much attention recently [6, 8, 9, 21, 22, 26, 28, 56] and

function execution latency, being the latest's greatest problem the need to get data from external storage. The FaaS solutions are characterized by having 2 specific layers, the computing infrastructure and the remote storage, the decoupling between these 2 layers give elasticity possibilities in detriment of lower latency.

During this subsection different papers with related works are analysed and conclusions about the implemented caching solution, hosting methods and trade-offs are taken. Finally there will be a table that resumes some of the main aspects for the implementation of the paper's solutions.

2.4.1 OFC: An Opportunistic Caching System for FaaS Platforms

Opportunistic FaaS Cache is a RAM based caching system that uses the over-provisioned memory to reduce latency without changes at the code level, or effort in configurations. Additionally and to avoid cold starts, the platform keeps the functions alive for several minutes, giving the possibility to aggregate these idle memory and provide a distributed caching system.

One of the questions raised in this paper is based on the scalability of the caching system that fits short function executions. One of the strategies, since the worker node will lack memory is to not cache data that is unlikely to be reused. The second strategy is based on a replication algorithm that keeps hot objects on active worker nodes. This solution exploits the overprovisioning of memory and reports improvements by up to 82 percent and 60 percent in the execution time of single-stage and pipelined functions accordingly.

When a function is triggered the choice of the worker that will attend that request is based on a Loadbalancer decision, this component maintains the available resources of all worker nodes, by using the function identifier and tenant, the index of the worker is computed, being this the preferred node to execute the function on it's "sandbox".

The "sandbox" is the environment that is kept alive to avoid cold starts, and they have three aspects that are important for a secure system:

1. A sandbox is never shared or reused between distinct functions or tenants;
2. It processes a single invocation at a given time;
3. To mitigate cold starts it is kept alive for some time to anticipate subsequent requests of the function that was previously executed;

The solution applied is based on the abundance of wasted memory that is applied at the provisioning of the environment that can be used to develop a distributed caching-system. The problem is caused by the cloud tenants that over dimension the memory resources that are configured on the sandboxes. The trend of over dimensioning the used memory can be followed because the same function can be triggered with different arguments which can lead to variations in memory needs. The second cause of memory resources waste is the mitigation of the cold start problem that would cause even more latency in each function execution, since a new environment would need to be provisioned for each trigger, which costs time. This type of solution is a double-edged sword, since the cold start problem is mitigated, but the memory resource is considered a waste after the first execution, since most of the times there are no subsequent requests for several minutes.

In the OFC project [65] the caching policy is defined to store objects that satisfy the condition of being smaller than 10 MB and that the predicted benefit of caching the object is significant enough to cache it. In addition and to reclaim needed space, the caching Agent discards objects that aren't accessed with a pre-determined frequency.

Another related project also identifies the previously mentioned problems on FaaS systems, being one of them the degradation of response times in systems that consume data from databases and file stores, that relates directly to the caching system solution studied for the thesis proposed solution development.

Main cloud providers, like AWS use caching systems such as Redis and Memcache through Amazon ElastiCache, but these solutions are external caches, causing the access to have the overhead of network calls. A proposed approach is to use the caching system as an internal component, even with the constraints that come with this approach, it is much more efficient in terms of latency. Some of the constraints are related to FaaS already identified problems, cold starts that are related to new sessions, since each session has it's own global variables declared, a new request will cause a new session to be instatiated, but every subsequent request will be attended by the previously deployed session if this hasn't been closed or suspended based on requests frequency. Therefore, every time the memory of the function container is used as a caching system, the cache can be invalidated if the requests frequency drops and the session is closed. This behaviour incurs in the conclusion that having an internal cache is much more useful in systems that have recurring calls to the same function.

EfficientFaaS has an internal and external cache in the GCP Functions Framework package. The trigger is routed in the express router, and depending on the caching

type that is defined on the configuration file, the functions manager will route the data call to the cache, otherwise, if the key isn't found in the cache data structure, the function is executed and returns the calculated to be set on the caching system. This value can be used to respond to subsequent requests that have the same content on the request object. The Opportunistic Caching System makes an estimation on the memory resources required by the functions and uses the remaining as a cache. It can be said that OFC uses an In-Process caching system, that is located on the function's execution process just like one of the possible configurations for EfficientFaaS.

The main contributions of our work to the problem of performance issues based on external data storage access is the development of a generic model that integrates with different caching systems for immutable data in order to reduce latency during FaaS function execution. Also, the transparency to change the type of caching system used based on a setup phase, that is done by adapting the configuration file definitions.

2.4.2 Pocket: Elastic Ephemeral Storage for Serverless Analytics

The communication of data between serverless tasks in a difficult challenge, so one of the approaches is to use an external data storage. As stated before, this type of solution impacts latency values, as a motivation it is developed an elastic distributed storage service that defines the correct size of storage clusters to provide better performance levels.

The principles applied to the solution are the separation of responsibilities, since three different components are defined, the control plane, metadata plane and data plane. The first one manages the size of each cluster and data placement inside those storage units, the second one tracks the data that is stored and the data plane contains the data. The second principle is the Sub-second response time where every I/O operation targets sub-millisecond latencies and has elasticity as a property mainly because the controller can scale the resources and provide load balancing techniques. Finally, the third principle is the Multi-tier storage where I/O demands are satisfied by different media storage devices, according to the needs (DRAM, Flash or disk).

In terms of Architecture, as previously referred the system contains a controller with one or more metadata servers, and several data plane storage servers. Being this component the brain of the solution, it provides scaling solutions as the requirements vary also making decisions about data placement inside the different nodes. For bigger objects, the system distributes smaller parts across the storage servers.

Comparing Pocket's solution to EfficientFaaS, it is seen that Pocket's uses different

remote storages focusing on minimizing the costs of using a paid caching service. EfficientFaaS uses redis cache as the remote caching system, being one of the Future Work ambitions to unlock the software to be easily adaptable to other caching solutions.

2.4.3 InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

The InfiniCache is an in-memory caching system that is deployed on serverless functions. As a proven thesis, large object caching improves the system's performance in terms of access speed in cluster computing, these objects are heavily reused and are accessed less frequently than smaller objects.

To know the distribution of object sizes, the author analyzed production traces from an IBM Docker registry, which was collected from two data centers, one located in London, United Kingdom, and the other one in Dallas, United States, from the registries, it was concluded that 20 percent of the objects are larger than 10 MB, concluding that large objects caching solutions have a great impact on system's latency values. However, since large objects occupy large amounts of space and this resource is limited there is a standoff between caching smaller or larger objects. As a simple solution to mitigate this choice, the managers could allocate memory resources to store objects of large sizes, but it can't be forgotten that this comes impacts the total cost paid to the cloud provider.

One appealing approach to the problem is to store objects in the function's memory, until it is claimed back by the cloud provider, is then inserted into a new function. The billing advantage is evident since the cache of an object would only be billed when there is a request to the function, since there is a "warm" period to mitigate "cold-starts", the object would be cached without impacting the bill value. The cloud providers have limitations for cloud resources usage, so this idea isn't viable since, for example, and AWS Lambda, has between 128MB and 3008MB of memory and allocates CPU based on the previously chosen value.

As a main difference between the developed project and the InfiniCache solution, the latest uses the memory of cloud functions to cache the objects, while the developed project makes use of the node [44] and redis [35] caching systems to store the key-value pairs.

2.4.4 Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure

The elasticity component is considered important, but it creates a challenge to implement the application as it is needed to move large amounts of data between different functions. Locus is a solution that combines slow and fast storing systems to achieve a good performance level. As a result, the cluster time in terms of cores/second is reduced by 59 percent and, even being 2 times slower than Amazon Redshift it doesn't require provisioning time, that is in the order of minutes on the Web services provider side.

Since containers are lightweight and can be scaled easily, the improvements in processing times are evident in workloads that require variation in the number of cores. Also, to overcome the latency caused by storage systems, Memcached or Redis are good solutions, that support higher request rates. However, these systems are expensive, making them a bad solution in terms of economic value. The read/write operations affect performance results and are variable based on parallelism and memory sizes.

In the Shuffling, Fast and Slow solution, there are 3 defined methods to study the overall performance, the slow, fast, and hybrid storage systems, that answer shuffling operation demands obtaining results based on these operations.

As a direct comparison between the developed project, and the Locus solution, the Locus focuses on elasticity and pricing, sacrificing the duration and total memory a function can use in order to process client requests. For the developed project, the main focus is to improve the performance of the GCP Functions Framework and give transparency to the user of the project.

2.4.5 Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing

Since the FaaS approach has a stateless nature, it forces that the states have to be maintained by an external data storage system which causes a degraded performance on requests processing. The serverless platforms also have the cold start problem that was previously explained, and a large memory footprint that limits scalability. In the paper [30] it is defended that there should be isolation abstraction which provides memory and resource isolation between different functions. But data should be co-located with functions, reducing data access times.

The Faaslets are described as a solution for serverless computing, which supports stateful functions with shared memory access. Each of these has an associated function and has a dedicated thread that can access a local or a global defined state since the memory regions are shared, avoiding isolation overheads. The Faaslets are also in a suspended state, so when they are launched the cold-start problem is mitigated since the instance was pre-initialized. The memory footprint of this solution is in the order of 200 KB and the initialization time is less than 10 ms.

Fasslets also include an API that supports chained function invocation, interaction with shared state containers, calls for memory management, and others. There is also concluded that containers contribute to the cold-start problem, and can be improved in order to mitigate it's impact. By recycling containers, isolation is sacrificed, but the initialization overhead is reduced.

By comparing the Faasm solution and the developed project, it is concluded that Faasm shares certain memory parts between functions on the same namespace, giving the possibility to functions that execute on different processes to access the same in-memory cached values. On the Functions Framework adapted solution, functions in different processes can only access the same storage for Out-of-Process or Network scenario. In-process scenario, the stored key-value pairs are on the process memory, so functions that run on a different process can't access the same storage.

2.4.6 Cloudburst: Stateful Functions-as-a-Service

The FaaS platforms as stated before are responsible for scaling resources in order to respond effectively to peaks of requests or load that involves processing. However in terms of deficits the storage services provided have high latency and are isolated with each other. Functions are isolated from each other, so they can't call each other since point to point invocations are disabled. And as a third point, the nested function call are slow.

In order to enable stateful serverless computing, it is proposed a logical disaggregation with physical colocation. Disaggregation is needed to effectively scale applications, and hot function's data should be kept physically close to reduce lantency in data I/O operations.

Cloudburst is built over Anna, that Is a low latency autoscaling key-value storage system and provides the benefits of commercial solutions, but trying to reduce the shortcomings. The performance bottlenecks can be addressed with local caching systems in every machine that executes function invocations. In terms of impacts it

could cause inconsistencies, since it is a distributed system, but it could be mitigated by solutions like the Quorum Paxos algorithm.

In terms of Programming, the processing occurs in the cloud, being the results sent directly to the client. It is provided an API that enables KVS interactions via get and put methods, enabling messaging between functions. The first function writes its ID, the second one waits that key to be populated and reads it's value, then uses the API to send a message and establish a communication system, via a deterministic map-ping mapping system that converts the Id into an IP-Port key value pair using .

In terms of architecture the autoscaling is independent of the Anna KVS system. There are four different components, function executors, caches, function schedulers and resource management systems. The scheduler receives client requests, routing them to the executors. The caches are located between each VM and the KVS system, and makes the most frequently used data available to each request. They are updated by the executors.

The executors, are independent processes that deserialize requests and has metadata that is provided to other components. Function schedulers define the execution of functions based on requests, by picking an executor and forwarding requests. They also track nodes that are saturated with requests and report the utilization levels, picking new nodes to process the requests when this situation occurs. The monitoring and resource management system, tracks the load and performance of the system to make scaling decisions.

The caching system creates a "snapshot" of the cached objects after the first read and sends the timestamped data when invoking a down-stream function. To maintain consistency the executor needs to read the same version of the variable.

2.4.7 FaaST: A Transparent Auto-Scaling Cache for Serverless Applications

It was previously stated, FaaS platforms rely on remote data storages that impact latency, in this paper, a solution is addressed to mitigate the problem of remote data access, an auto-scaling distributed cache making the access local. As a first point, it is needed to understand that the frequency of the function's invocation has variations, so the ones that are rarely invoked shouldn't have their data cached, since the memory resources would be wasted. However, not caching data at all is also a bad choice, producing a worst performance.

Another important point is based on the data size, if it is stable or varies since, with objects that are smaller than the caching space, the solution is simple, otherwise the cache resources need to be scaled to accommodate bigger objects. Finally, another worry is the lack of transparency that caching services have to users, because sometimes the access is provided via a separate API. These deficits are proposed to be tackled by the solution described in the paper, the in-memory caching system. By having each application with a dedicated cache, instead of an external caching service, it is expected to have a reduction in data access times. Being also possible to pre-fetch the most popular data and ore-warming of applications it improves the efficiency of the overall system's response times.

The scaling politics are based on the object's size to increase the bandwidth and the frequency of object accesses to increase the overall caching size.

Another important factor is the "intelligence" in the scaling process, while Pocket, InfiniCache and Locus dynamically scale, based on the amount of load, the OFC scales based on predicted memory usage.

The FaaST design and architecture, are based on transparency and an auto-scaling caching system. The data is persisted only during the application's lifetime not going beyond that time frame. The scaling politics are based on the frequency of invocations, data reuse patterns and on the bandwidth scaling which are benefic for large objects.

Looking at the architecture particularities, each application has a cachelet being supported by a cooperating distributed cache. The shared cache is considered a problem, and not a solution based on the difficulty of implementing a management system that denies multiplexing of applications accesses and I/O operations on the same cache.

To improve latency results, this solution also Pre-warms data, avoiding an unnecessary request to the external data storage during the function execution. In terms of performance, the pre-warming technique improves performance by 74 percent over cold-starts, this only occurs when the application is not executing or right after a function execution to avoid impacting on-demand accesses. As known the memory capacity is limited by the provider, so the caching system is limited by this resource incurring in the need to evict cached data. The method applied evicts data when the memory consumed by the function and the cached objects is within a small percentage of the total application's memory. There are multiple eviction policies, the first one evicts objects that are not owned by the evicting cachelet, then it chooses the Least-Recently-Used objects. Another policy targets large objects that are

not owned by the evicting cachelet. Finally, before resorting to the Least-Recently-Used objects, it evicts the ones that are larger than the pre-defined size threshold.

Both EfficientFaas and FaaST solutions benefit from remote and local caching services, creating a transparent solution for the user. As a big advantage of FaaST, it uses an RPC solution to exchange memory addresses between the application function and the cachelets, in order to share the memory with stored data between different processes, which doesn't happen in the In-Process EfficientFaaS solution, where each process has it's own caching node-cache instance.

The upcoming table represents a summary of the main conclusions taken from the studied related work. Pocket paper states that it resorts to storage nodes as the go-to solution, those are hosted in VM's. In the InfiniChache case, the objects are cached on the lambda function (in process). The Locus solution uses a storage mechanism that has to be provisioned by the user, like ElastiCache from Amazon. The Faasm uses Anna, a key value storage that can be instantiated on containers or virtual machines. The case of Cloudburst uses a local cache per function execution. Finally, OFC and FAAST, both use the invoked function's memory.

	Caching Solution	Host	Trade-offs
Pocket [24]	Storage Nodes in servers using different storage media (Dram, Flash, disk)	VM de Serviço Cloud AWS	Non-Transparent, need to be provisioned
InfiniCache [27]	Lambda cache pool with cached objects on the lambda function (In-Process)	AWS Lambda	No cache size scaling, no objects pre-warming
Locus [25]	Storage mechanisms provisioned by the user, for example Amazon ElastiCache [1.4 Challenge 2]	Aws Redis (Elasti-Cache)	Non-Transparent, need to be provisioned;
Faasm [30]	Anna (KVS)	Containers or VM	Non-Transparent, need use external API. Anna
Cloudburst [52]	Local cache per function execution VM, storing data in key-value stores	Local VM	Non-Transparent, need use external API
OFC [65]	Per-Worker cache using the overprovisioned memory (In-Process)	RAMCloud server	No objects pre-warming
FAA$T [53]	Store on the Invoked function's memory	Kubernetes	

Table 2.1: Related Works summary

3

Proposed Solution

The present chapter starts by introducing the architecture of the chosen framework and the adaptations that it suffered to include a caching solution. The main goal of this implementation is to reduce the function's execution times by consulting a caching system before making any processing.

Gains of this implementation vary based on the function's processing that has to be done. If it is a function that runs a simple and light process the gains can be small, however if the job is executed for example on large images or heavy video processing [7, 20], the gains can be more significant.

3.1 Architecture FaaS

In this subsection the generic FaaS architecture with and without caching services will be explained in further detail. On Figure 3.1 a typical framework that provides FaaS infrastructure has 3 main modules [76], 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 events to function triggers that start the instance of the function. Finally, the function mapper makes the mapping to the runtime environment. A request is received by the API and the parameters are passed to the interceptor, which as a first approach, based on the functionName parameter, forwards the request to the function instance that is being executed on the runtime environment. The instance has access to external APIs

or external storage systems, by using the cache provider instance, and returns the response to the client based on the code that the developer set the function to run.

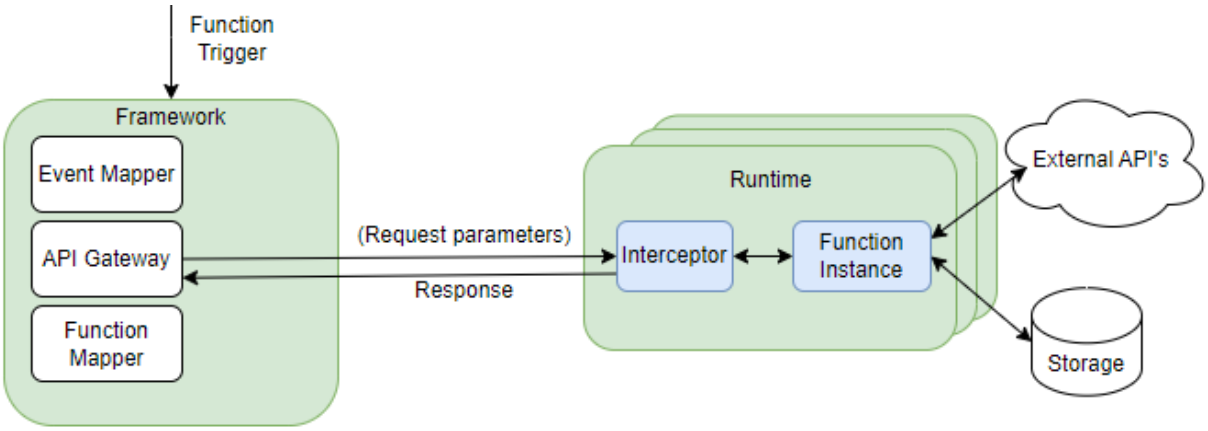

Figure 3.1: FaaS architecture - without cache

As for Figure 3.2, on the left side of the image we can see the components that are transverse to every FaaS solution and on the right side it is represented the building blocks of one of the main contributions made. The incorporated caching service used during run time is composed by a couple of modules. Additional modules are the Configurations, and Cache Proxy that provide the possibility to use the caching service.

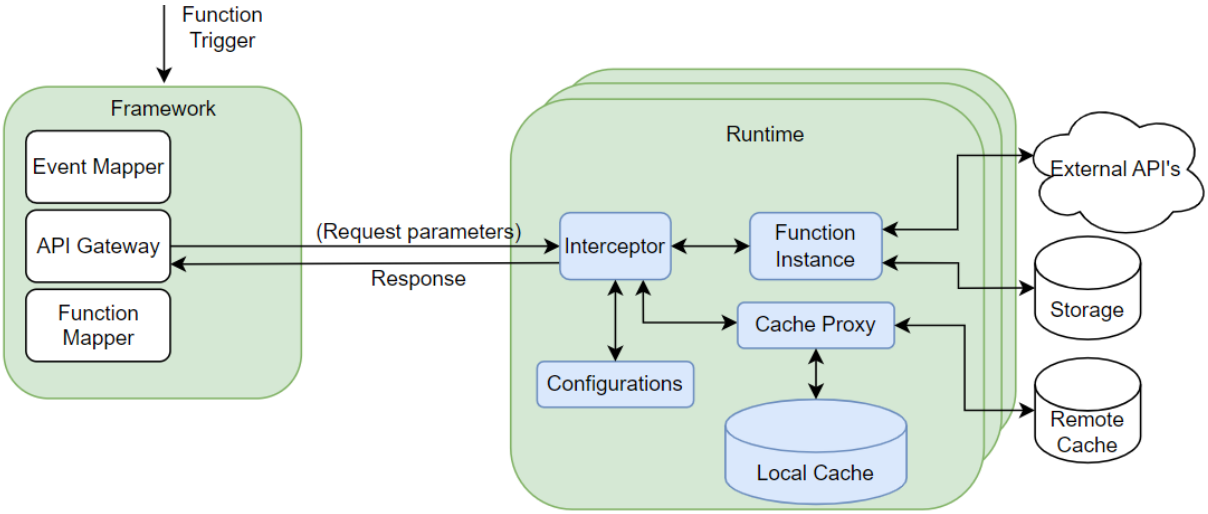

Figure 3.2: FaaS architecture - with cache

In this case, the interceptor also connects to one of the caching systems, it connects to the local or the remote cache based on the type of test and with the configurations provided by the Configurations Module. This connection generates a Cache proxy instance that is used along the program to get and set key/value pairs on the caching system, making these calls to the local or remote cache based on the instantiation that occurred on the Interceptor module.

3.2 Block instance diagram Functions Framework

The Functions Framework building blocks depicted in Figure 3.3 has some nuances compared to other Frameworks. The Main module receives the arguments passed by the user and starts the instatiation of the function. Next the `Loader` module has the function `getFunctionModulePath` called receiving the code path (1). On the same module the User Function is obtained using the `functionTarget`, one of the parameters received by the `Main` module and passed by the user. The `Function wrappers` has the task to wrap the user function based on the type of event that triggers it, if it is an http event it calls the `wrapHttpFunction`, if it is a cloud event it calls the `wrapCloudEventFunctionWithCallback` function, if it is another event type, it calls the `wrapEventFunctionWithCallback`. The `Server` module is called, registering in the middleware that the requests with the trigger type to the function calls the defined function. The server is returned to the `Main` module (2) and the `Invoker` is called (3) where the server is registered and the events of possible Exceptions are registered to be triggered if an error occurs.

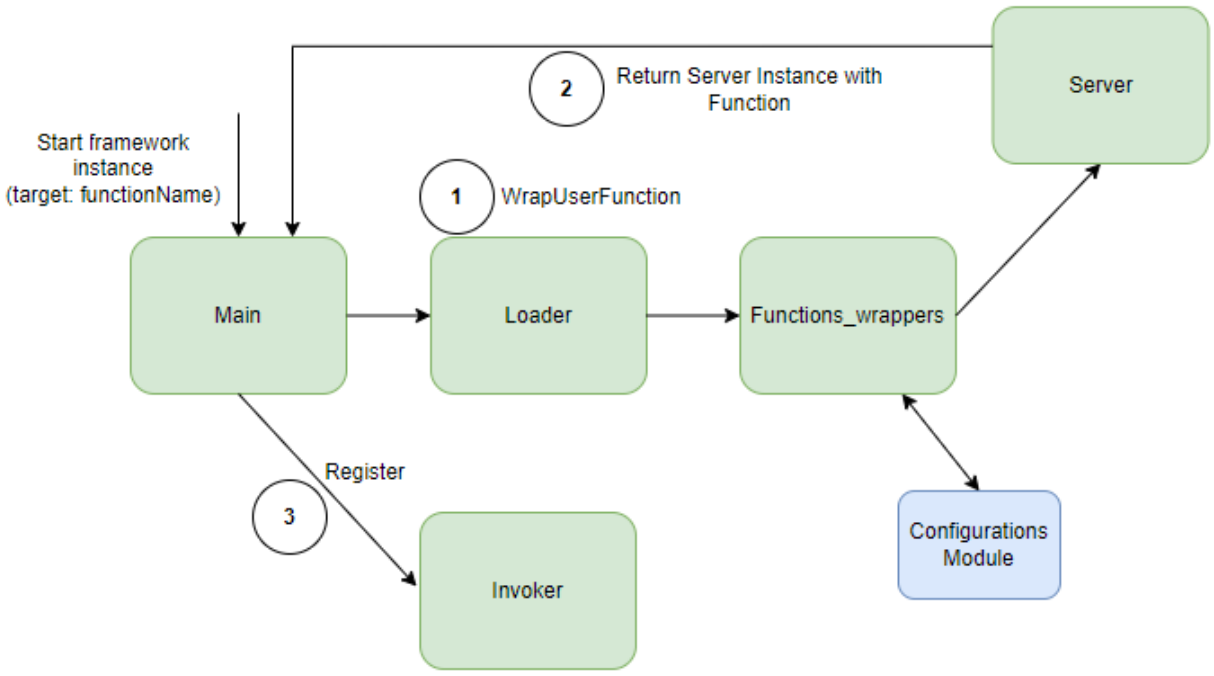

Figure 3.3: Function Framework building blocks

3.3 Functions Framework with / without cache

In this section, there are 2 diagrams, one of the function execution flow without a caching service, and the other with a caching service, which can be one of the three previously explained types, In-Process, Out-of-Process, or Network.

On Figure 3.4, it is seen that the request is processed by the Server, sent to the `Functions wrappers` module where it's middleware or interceptor analyses the parameter received as functionName and determines the function to be executed on the `Functions Manager`. In this module, the instance of the function is invoked, on the `Index` module and the result is returned to the Server that uses it on the response to the client.

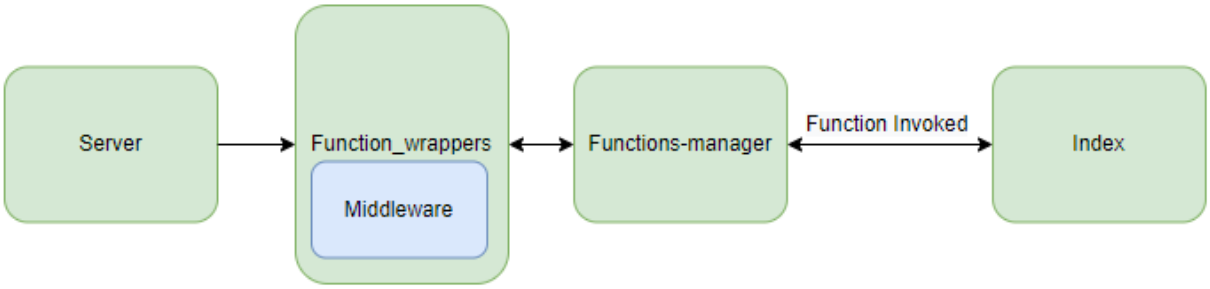

Figure 3.4: Functions Framework Execution flow without cache

As for Figure 3.5 the scenario includes the caching service mechanism on the Functions Framework modules. In this case the server also provides the parameters of the request to the `Function Wrappers` that has the Interceptor or a more complex middleware waiting for a request to be received. The interceptor then, based on the caching service to be used (which is determined by the function invoked), obtains the cache configuration parameters from the Configurations module and instantiates a cache proxy instance that is passed to subsequent modules.

The `Function Wrappers` passes the cache proxy instance and the calculated hash of the relevant parameter to the `Functions Manager` that queries the local or remote cache in order to return the result to the client or continue the flow. If the hash is already cached, it means that the same request was previously attended and since the response is the same, it returns the cached valued to the client, if not the `cache proxy` is passed to the function instance on the `Index` module. On the Index module the function is executed and the key-value pair is set on the local or remote cache. Finally the result is returned to the client.

With the analysis of the `Functions Framework`, the modules that had to be adapted were identified and we advanced with the development of a proposed

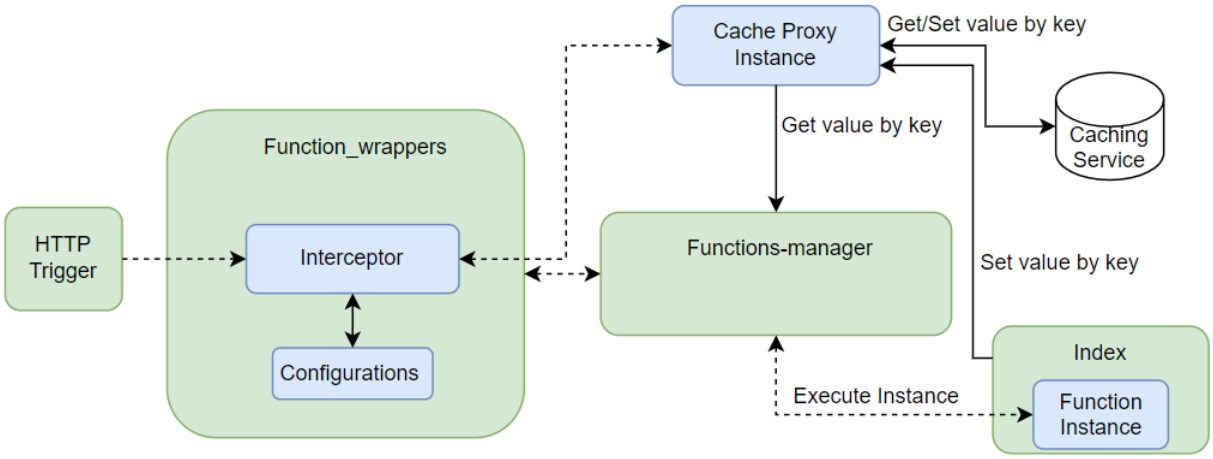

Figure 3.5: Functions Framework Execution flow with cache

solution. Chapter 4 details the Implementation based on the proposed solution, along with the designed use cases.

4

Implementation

In this chapter, the details of the implemented solution are discussed and explained in order to better understand the components that were modified and the ones that were included in the new Google Functions Framework.

These details include the dependencies of the project, composing modules, and the three caching types implementation.

4.1 Proposed Solution - Extending FaaS GCP

The extension of FaaS GCP frameworks consists in accommodating a generic and dynamic caching model that is consulted before the execution of the function with the goal of reducing the response times, by returning already processed values. The middleware works with different types of cache that can be chosen by configuration.

For the three scenarios referenced on Chapter 3, the first one to be developed is the in process caching system where the cache is an object which is accessed inside the application process, so, if the application is stopped, the caching system is invalidated, since each function instance has it's own cache.

With the `cachingService` string, the decision of connecting to the node-cache [64] (local caching service) or the Redis cache (remote caching service) can be made, and the connection string used defines if the Redis connection occurs to the same machine instance or the network instance. The request object is used to populate the property

client which has the instance of the cache client from which the functions set and get can be invoked.

With the goal of demonstrating the benefits of our approach using a common use case, we choose the thumbnail generation scenario, based on the referenced use cases of the paper "Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends" [77].

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.

One of the concerns in the development of this solution is the transparency for the programmer [4]. 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 define what's the property name that contains the content of the request. This approach is based on techniques used in aspect-oriented programming [3], where behavior is added to existing code, without explicit code modification. Another aspect is the process of caching results, where the programmer's code only has to return the object to be cached, since the infrastructure takes care of getting and setting the key/value pairs, following the commitment of maintaining the framework transparent and independent of the function's code.

The properties of the configuration file are detailed in 4.1. 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, the node or Redis cache. If it is the Redis remote caching system, the redisRemoteConnectionString property is used to define the connection string to the caching service. Finally, the testCached property was used during the latency calculation tests, in order to define that the cache wasn't used for tests that were calculating the latency of the requests without using the caching services.

```
{
  "property": "image",
  "cachingService": "nodeLocal",
  "redisRemoteConnectionString": "redis://default:tfm02-
   ↪ redispassword@34.105.181.115:6379/0"
}
```

Listing 4.1: Configuration File

4.2 Implementation Details

The Functions Framework package from Google Cloud Platform can be found on the official github repository [59]. In addition to the code of this framework, EfficientFaaS [70] also depends on the following packages:

- 1. Crypto [72], a module used to generate the hash keys for the cache Key-Value pairs that are stored;
- 2. Image-Thumbnail [71] 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 [66], an in-memory caching system used to store key-value pairs.
- 4. Redis [69] is an in-memory data structure that can be used as a database or caching system.

4.2.1 Function Wrappers - Interceptor

Handling the decision of using an In-process, Out-of-Process, or Network caching service occurs on the Function wrappers module where the implementation of the Interceptor is made. Based on the `cachingService` argument that is configured by the user on the settings file.

With the relation between the `cachingService` string and the cache used, the `Cache Proxy` instantiation is based on the setting from the configuration file. The `Cache Proxy` handles the connection as seen in listing 4.3, and communication between the developed solution and the caching service. The proxy instance is passed to the `functions manager` by the `function wrappers`, as seen in 4.2, which

passes it further along to the function instance. Depending on the type of cache to be used (defined on the configuration file), the connection is made to a node-cache instance or to a Redis caching service that is running on the same machine for the Out-of-Process test case or in a different machine on the Network for the Network testing case.

```
const wrapHttpFunction = (execute) => {
  return (req, res) => {
    const d = domain.create();
      // Catch unhandled errors originating from this request.
      d.on('error', err => {
        if (res.locals.functionExecutionFinished) {
          console.error(`Exception from a finished function: ${err}`);
        }
        else {
          res.locals.functionExecutionFinished = true;
          (0, logger_1.sendCrashResponse)({ err, res });
        }
    });

    var functionResult
      d.run(async () => {
        process.nextTick(async () => {
          req.body[propertyAccessTest.property] =
            req.body[propertyAccessTest.property]
              .toString()
              .replace("", "")

          req.hash_key = crypto.createHash('sha256')
            .update(req.body[propertyAccessTest.property].toString())
            .digest('hex');

          req.cachingClient = await cache_proxy
            .ConnectCache(req, propertyAccessTest.cachingService)

          functionResult = await functions_manager
            .FirstFunctionExecution(execute, req, res, cache_proxy)
        });
    });
    res.status(200).send("Ok")
  };
};
```

Listing 4.2: Function Wrappers

As seen in listing 4.3 the caching options are nodeLocal, redisLocal or redisRemote, being the default case used the connection to the redisLocal, here the code could be simplified by deleting the redisLocal case. The decision of keeping it, was in order to ease the readability, by having the explicit case. The connection on `ConnectCache` function only occurs if the `req.cachingClient` property is null, it 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.

The `Cache Proxy` instance is passed along the flow of the program, is used to get or set the cached values based on its keys.

```
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 4.3: Cache connection details

In the module, it is used a JSON file, the `ObjectProperty.json`, that defines the name of the property that has the content, in this case the image content, that is

received on an HTTP request received by the server and passed to the interceptor.

When the request object is received on the interceptor, the object is passed to a function on the Cache Proxy that handles the generation of the Key. The Key is generated with the sha256 [75] hash function for the base64 value of the image, generating a unique key per image.

The key, Cache Proxy and the request object are then passed to the Functions Manager module.

4.2.2 Functions Manager

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 that is eventually returned to the client that originated the request. Otherwise, if the key isn't found on the cache, the function is executed, on our case the image *fingerprint* is generated and set on the cache by the `Cache Proxy` instance, finally returning the calculated value to the client. On the user function, the value should be returned by the function in order to be set on the cache.

This second case, provides no advantages in terms of latency, since the value isn't cached, and can't be returned immediately to the client, but subsequent requests with the same content will benefit from the implementation and developments.

4.2.3 In-Process Implementation

In-Process case, as previously explained is dependent on the node-cache module, 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. For this reason, the cached values aren't shared along multiple instances of the application, not giving the scalability power that the Network caching service provides.

As for the implementation, the interceptor uses the Cache Proxy module to connect to the cache, via the `ConnectCache` function, which receives the request object and the configuration `cachingService` string.

The node-cache [66], 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, then calls the functions manager module that with the Cache Proxy received as a parameter, invokes the GetValueByKey method, which with the pre-calculated hash value of the image, it can access the specific cache, and check if the value is cached to be returned. If it isn't cached, it calls the execute callback, which triggers the execution of the function, passing the request and response object and the Cache Proxy instance. On the function, it sets the key-pair value on the respective caching system. Otherwise, if the value is cached, it is returned to the client.

4.2.4 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 in this case, multiple instances of the function can access the cache on that machine. This means that in terms of scalability, only a vertical scenario is possible, where more resources are allocated to the same machine since the caching service isn't shared along different machines.

Redis [69], the caching service used in the testing scenarios, is an open-source in-memory data structure that can be used as a caching service, message broker, streaming engine, or database. 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.

4.2.5 Network Implementation

For the Network case, the caching service system runs on the network on a different machine from the one where the function is being executed. This means the machine can be accessed by multiple different machines running the same function, and accessing the same cache, with values that could've 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 [2, 13–15].

Being a case that is like the Out-of-Process case, in terms of the type of cache, the major difference is also on the connection process that occurs on the Cache Proxy module.

The next steps are the evaluation of results for the proposed scenarios with a brief description of the setup that is discussed in Chapter 5.

5

Evaluation

The development of the three previously described use cases on top of the FaaS Functions framework brought the possibility of having a caching service directly coupled to the solution without further configuration or additional developments, which brings performance advantages that needed to be proven by testing each one of the use cases.

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, which brings performance advantages since the process that needs to access it is exactly the same, causing low latency values when a client wants the thumbnail of an already processed image. This case theoretically has the lowest latency values in cache access from the three different cases which needs to be proven on the next phases.

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 [35] 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.

Finally, the Network use case will be tested by using different virtual machines on the Google Cloud Platform, where the function process runs on a process inside a virtual machine and the caching service is running on a different virtual machine. The scalability of this third use case is incomparable to the previous ones, in view of the

fact that the Redis cache is completely isolated from the function's running process. In a real scenario, 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 accessed by different clients could have already been processed, obtaining the result much faster by accessing the cached thumbnail value.

The main goals of the evaluation are:

1. Evaluate the time it takes to process the thumbnails versus the time to access caches in different locations, In-Process, Out-of-Process on the same machine, and on a different machine but on the same network.

2. That the latency to access the cached resources grows with the distance to the function's code, but the cost of doing so are outweighed by the benefits of avoiding the computation of the thumbnail, especially as the image grows in size.

3. In-Process cache isn't the ideal solution, since it enters the saturation point sooner than in the two other cases. Demonstrates that there is an added value to the system in using a cache that can be shared with different instances of the function.

5.1 Evaluation Setup

In the following sections, the evaluation of results will categorize each one of the use cases and bring the advantages and disadvantages of each one of them. As for an initial test of the use cases, the latency values of image processing, and time to obtain previously cached values from images were retrieved and compared. This test was made using three different images, that scale up in size, in order to better understand the impacts it creates on the functions-framework system.

The image sizes vary from 18Kb, 200Kb to a total of 4Mb on the latency test scenario for each one of the use cases. It is expected for the processing time to grow in proportion to the image size, the caching system a greater advantage for higher images than for smaller ones, since the order of magnitude between the processing time of a 4Mb image and the latency to access the caching system is bigger than cases where the image has low size.

The first tests were done with an e2-micro [73] from the Google Cloud Platform, a micro machine type with 0.25 VCPU and a total of 1 GB memory. The machine with the lowest specs was selected in order to easily overload it with HTTP requests, increasing

the number of requests until there are no more hardware resources, causing a decay in the number of attended requests.

On a different Virtual Machine, there is an instance of a docker container running the Redis server that stores the cached data that is populated and consulted by the function. This instance runs on a e2-medium, 0.5 vCPU, and 2 GB memory.

Apache JMeter application [46] was used to perform the HTTP requests. JMeter is an open-source software [62] designed to measure performances of applications and load test them [36]. This platform has a variety of possible configurations that gives versatility to our tests, mainly we will resort to threads, creating a simulation of multiple clients generating parallel requests [33] to find the saturation point of the machine in each of the cases. The application has a simple user interface where the model of requests can be configured and can be executed. However, for a viable load test, the command line mode was used since the GUI mode decreases the JMeter's capabilities. With this, every test was configured via GUI, and executed via CLI mode. It is expected that a saturation point is found during these tests, where the resources of the machine running the framework can't handle the number of requests to attend, so the request won't have an acceptable response time, being the acceptable time window for the ones obtained during the latency tests. In order to fine-tune the testing, and obtain reliable results in every testing there is a 10 second timeout window, where if the function doesn't return any response a timeout is counted by the JMeter application.

In each of the next sections, each use case will have it's results evaluated and further explained with the goal of understanding the advantages that one case brings over the other, and the impact on latency values when processing smaller or bigger images.

5.2 Small Image Test

As previously indicated, the first test evaluates the performance at the level of latency values. These values will be 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. The value calculation varies for which one of the cases if the test is without cache or any of the use cases with the cached thumbnail in Table 5.1.

Table 5.1 is composed of 5 columns, the first one states the image size in bytes, and the second one is the latency values when the image thumbnail isn't cached. As for the

	ImageSize (B)	Without Cache (ms)	Cache In Process (ms)	Out of Process Cache (Redis same VM) (ms)	Network Cached (ms)
	18617	39.112447	0.159357995	0.70927301	27.86512
	18617	36.35944	0.156939998	0.903744012	31.120851
	18617	36.754914	0.210119992	0.540549994	27.771786
	18617	40.678898	0.171569988	0.521357	27.79057
	18617	36.636939	0.36965701	1.06328699	26.825163
	18617	36.836955	0.155626997	0.708052993	24.243784
	18617	35.661015	0.11693801	0.543161005	24.013225
	18617	36.9172	0.121660009	0.853395998	30.477425
	18617	36.607837	0.143203005	0.482778996	29.599634
	18617	39.775737	0.217013001	0.473301008	23.942831
Average		36.7959345	0.158148997	0.625606999	27.781178
Standard Deviation		0.331645001	0.028827503	0.117986001	1.9611445

Table 5.1: 18Kb image processing latencies table

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.

As expected, the In-Process cache access latency is the smaller one, having the Out-Of-Process use case 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. In conclusion, the closer the cache is to the image processing function instance, 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.

The following subsections represent stress tests along each of the use cases. The tests were made using Apache JMeter as the client that invokes the functions framework instance that is running on the GCP virtual machine. In each of the cases, the cache wasn't accessed with the objective of quantifying the average number of requests with a valid response per second (Hits per second).

5.2.1 In-Process stress test

In Figure 5.1 it is seen the interaction between the Function instance and the cache occurs during the execution of an In-Process use case. As the name indicates, both of the elements are in the same process, causing an effect where different function instances have a directly associated cache, that isn't shared with other instances.

Part of the next evaluation tries to represent the saturation point of the In-Process use case, where a maximum number of requests per second will be hit.

In Figure 5.2, it was used 1 thread and a total of 500 requests were made. As seen, the x axis is the absolute time in the format hour:minute:second, and the y axis the number of hits per second. The number of hits stabilizes at 16 hits/second where it is maintained consistent until the end of the test. This means that 1 client isn't enough to

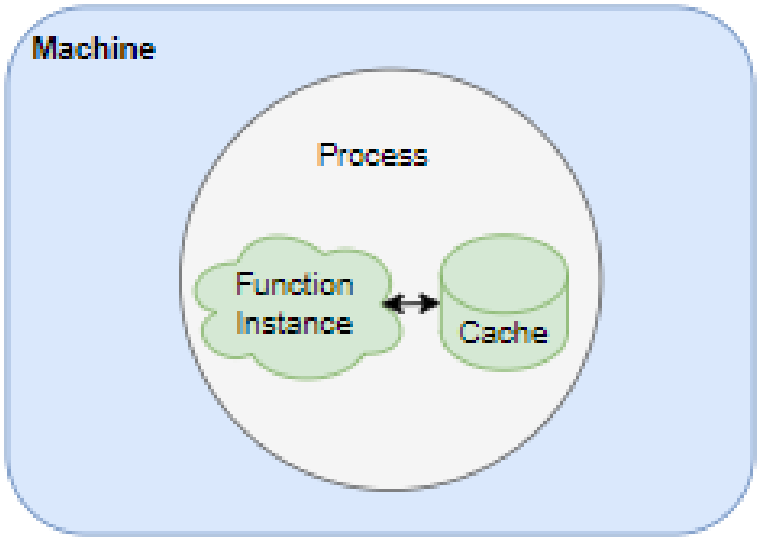

Figure 5.1: In-Process use case diagram

saturate the virtual machine, so the test has to be done with a larger number of clients, in this case, threads.

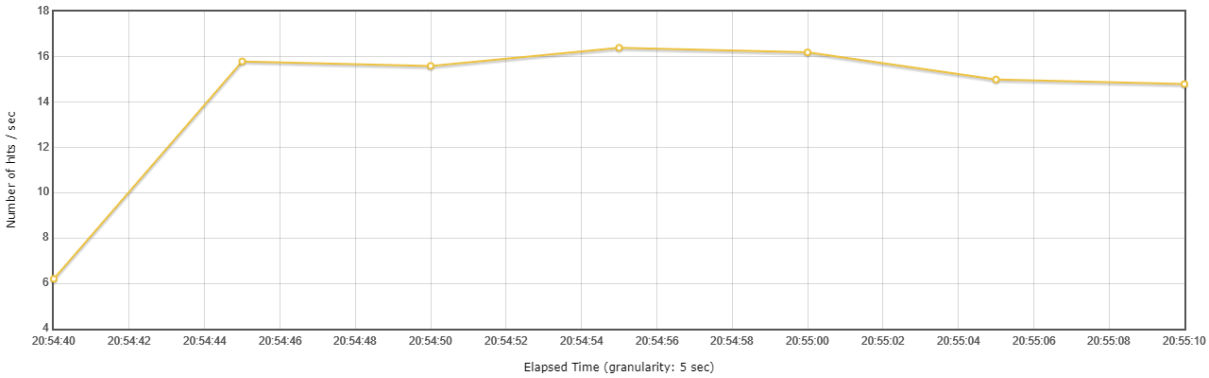

Figure 5.2: In-Process test 1 threads 500 requests

The increment of threads will be from 1 to 5 with the same 500 requests, but now each thread will be making a total of 500 requests, obtaining 500 responses, and then stopping until a maximum of 180 seconds of testing time. So if the requests don't have a response before these 180 seconds, the thread will be safely stopped after the pending requests have a valid response.

In Figure 5.3 the number of hits reaches a maximum of 80.60 and an average between 65 and 75 hits per second.

In this next case Figure 5.4 it is seen that a saturation point is reached after 30 seconds of testing which was the total testing time of the first case Figure 5.2, so after 30 seconds the first thread has finished, but 4 threads are still making requests. A maximum point

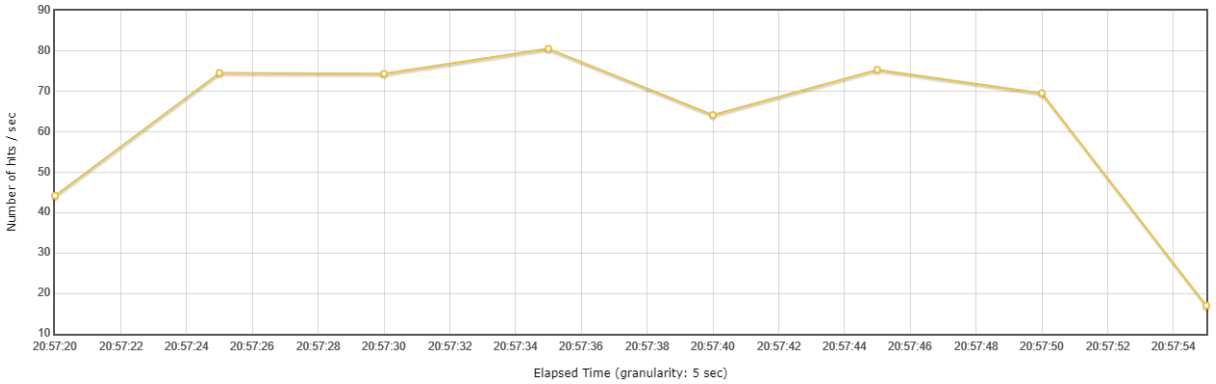

Figure 5.3: In-Process test 5 threads 500 requests

of 79.40 hits per second is reached maintaining the number of hits between 70 and 80, but decaying to an average of 20 hits per second, where now the processing only occurs when the virtual machine resources from other pending requests are released.

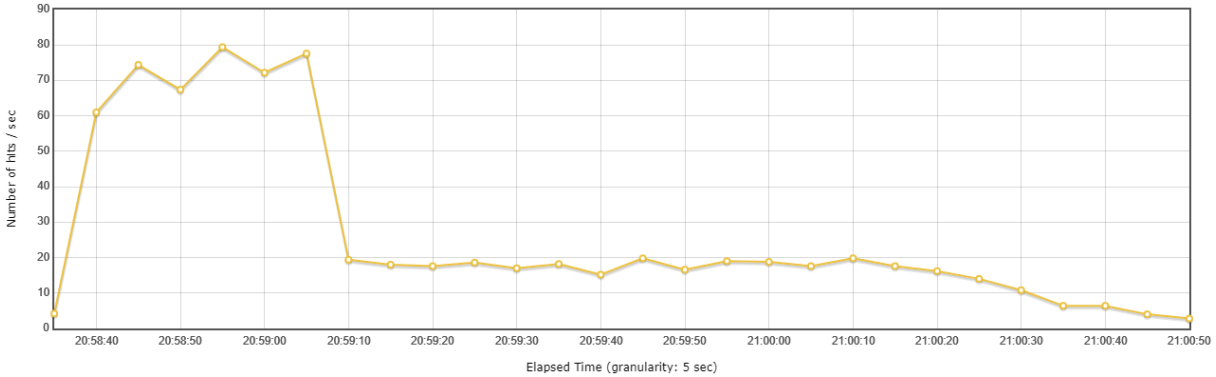

Figure 5.4: In-Process test 5 threads 750 requests

As for the next case, there is an attempt of over saturate the system, with 1500 requests for each thread. In Figure 5.5 it is seen that the peak is reached faster than the previous cases, to a total of 76 hits per second in 10 seconds and dropping from the saturation point to an average of 17-18 hits per second, nearly 1 hit less than the previous scenario.

It can be concluded that an In-Process cache wouldn't be enough for a real use case with significant load, so outside of the process or outside of the machine cache has to be used, since it is a shared cached across multiple instances of the function where the advantages are evident [2], as we can see on the next sections.

5.2.2 Out-of-Process stress test

In Figure 5.6 the interaction between the Function instance and cache is seen, where each one of them is executing on their respective Process. In this case, different

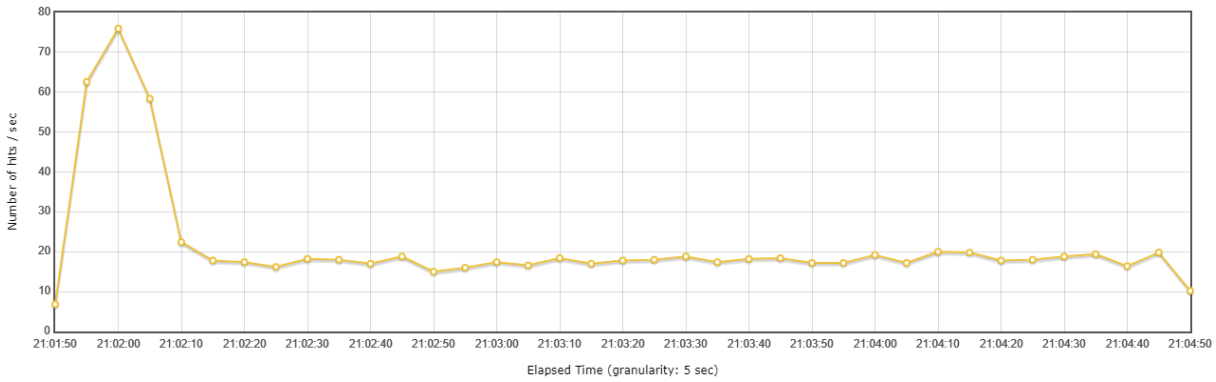

Figure 5.5: In-Process test 5 threads 1500 requests

Function Instances can access a single cache, bringing the benefit that a request that has been processed on a different instance, can have a response that will be consulted by a different instance, speeding up the process of attending clients that make requests that have previously been attended.

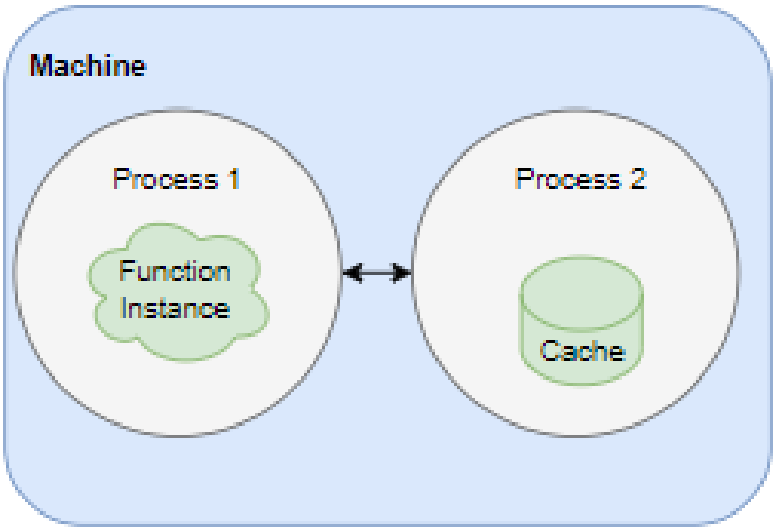

Figure 5.6: Out-Of-Process use case diagram

Out-Of-Process scenario is tested using a Redis instance on the same virtual machine where the functions-framework is running. This comparison between the In-Process, Out-Of-Process, and Network will also give an understanding of how the results change depending on the proximity of the functions-framework instance to the caching system.

In Figure 5.7 we can see that the pattern of the graph is comparable to the In-Process 1 Thread 500 requests case in Figure 5.2, providing the conclusion that in this scenario, one thread only isn't enough to make the virtual machine reach its saturation point.

For Figure 5.8, and compared to Figure 5.7, the biggest difference is the number of hits

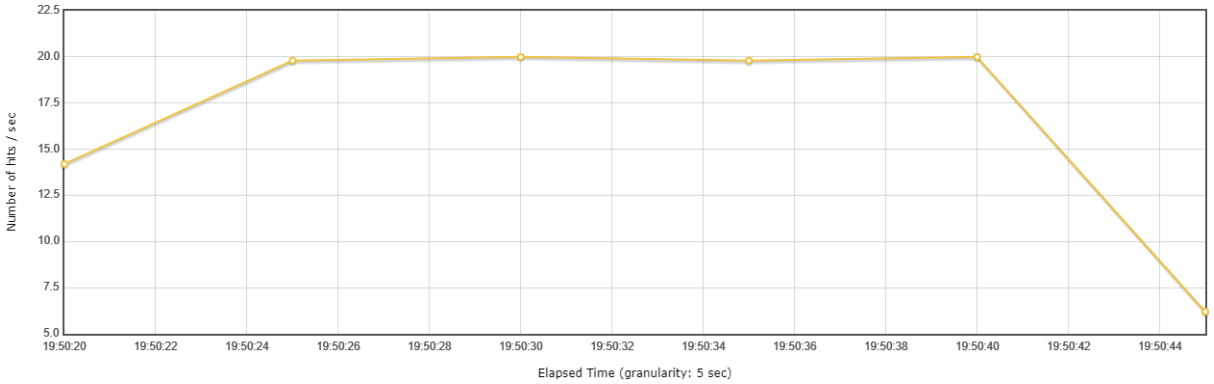

Figure 5.7: Out-Of-Process test 1 threads 500 requests

per second. This value scales up to nearly 4 times more hits, caused by the number of threads that go from 1 to 5 during the test.

Adding threads creates more parallel requests, if the framework can handle those requests, the number of hits will be higher, until the system saturates.

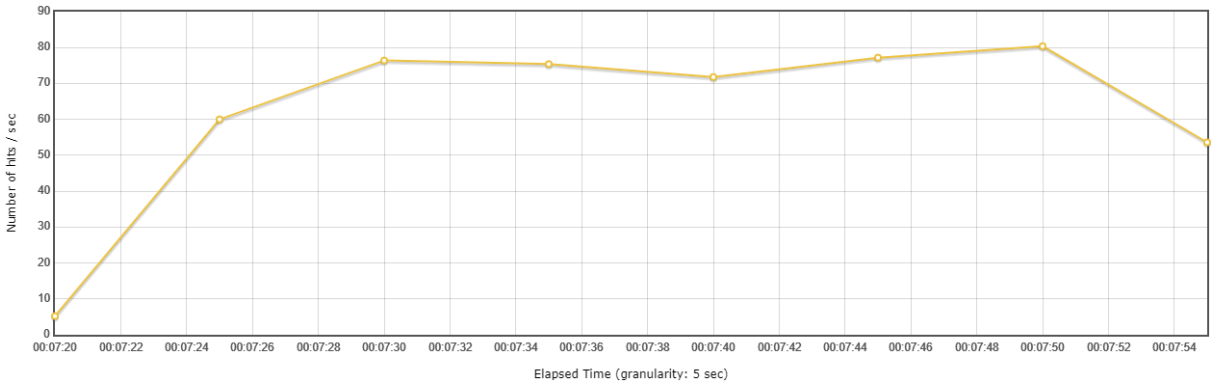

Figure 5.8: Out-Of-Process test 5 threads 500 requests

In Figure 5.9 it is seen that a saturation point is reached after 10 seconds of testing, with a peak of 75 hits per second, that comes down to a mean of 16 requests attended per second after a couple of seconds.

For Figure 5.10 the result is comparable to Figure 5.9, where the saturation point is hit, and after this, the hits per second come crashing down to 5-6 times fewer requests attended per second since there are no available resources to process the requests.

In conclusion, since the caching system is further away than the In-Process use case, the number of hits per second are lower than in the previously tested scenario, giving the conclusion that the access to the cache also takes time and resources from the machine.

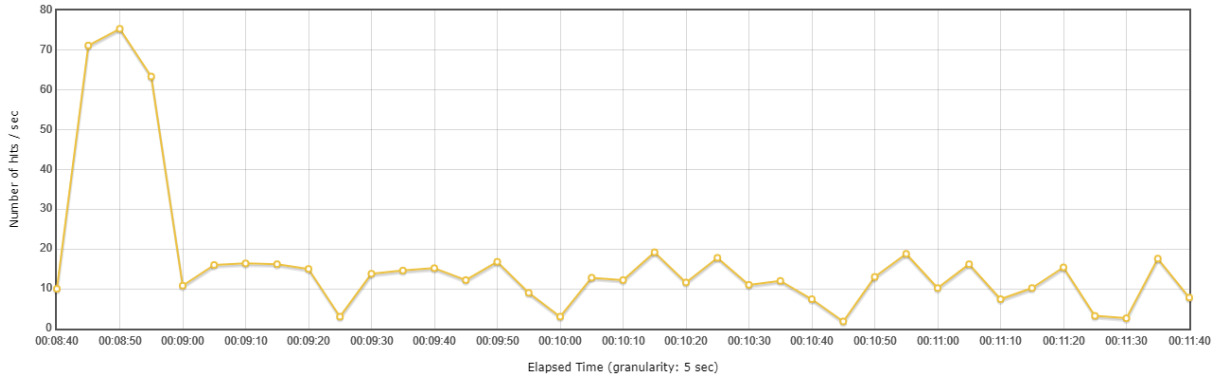

Figure 5.9: Out-Of-Process test 5 threads 750 requests

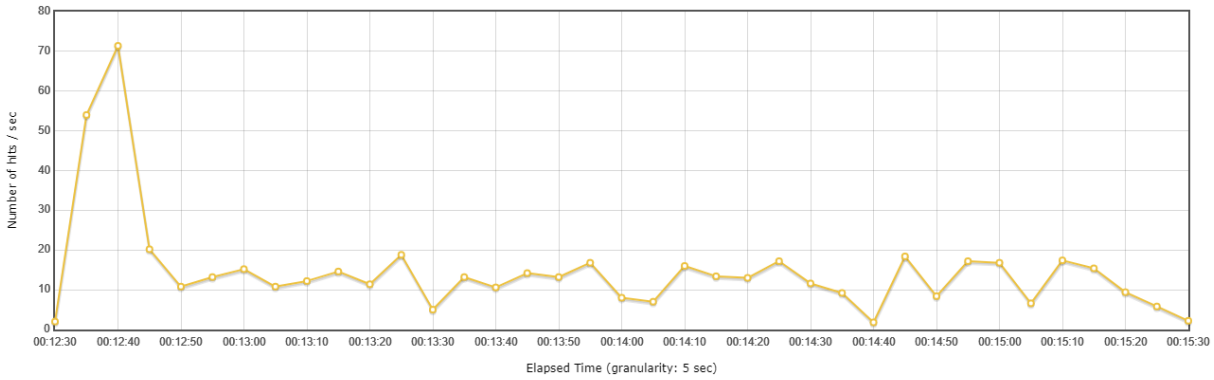

Figure 5.10: Out-Of-Process test 5 threads 1500 requests

5.2.3 Network stress test

As for the Network use case, there are two virtual machines, 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 5.11. The scenario gives the possibility of multiple virtual machines to consult the same data storage.

The first test using JMeter for this use case resorts to 5 threads making requests with a ramp up of 5 seconds, until a total of 2500 requests are made Figure 5.13. Basically, there is 1 thread running making requests, and every 5 seconds another thread starts to make requests until a maximum of 5 threads are making parallel requests.

As we can see in the image above the number of requests with a response goes up until there are no more requests, and then comes back down, so we conclude that the machine didn't get saturated with these configurations.

For the next case it was made a test with the same configurations, but with 750 requests per thread, so a total of 3750 to find the point where the requests eventually saturate the machine Figure 5.14.

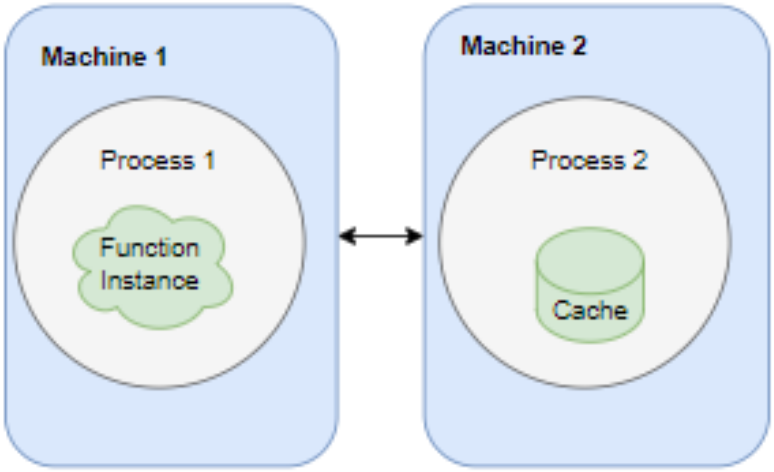

Figure 5.11: Network use case diagram

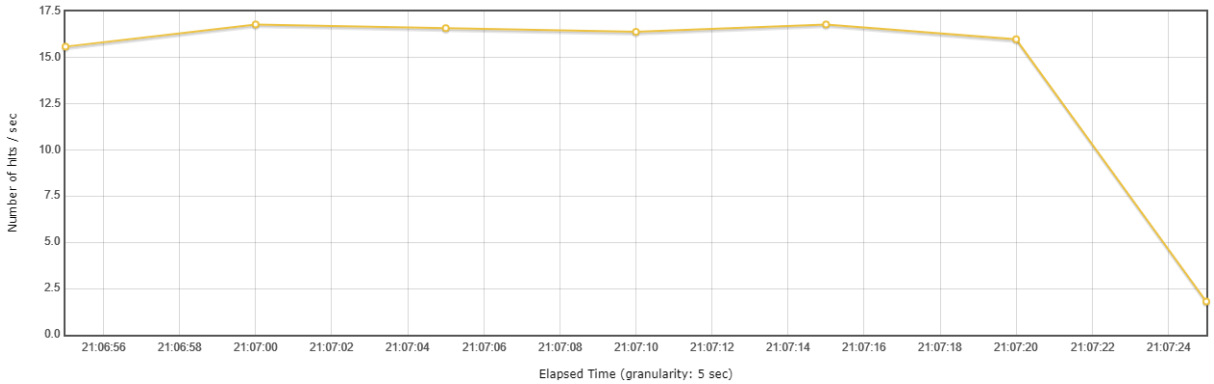

Figure 5.12: Network test 1 thread 500 requests

As for the last test, the properties were tweaked to 5 threads, ramp up of 5 seconds, but a total of 7500 requests in order to try and get the case where the machine is saturated with requests.

In this case, connection timeout's started to occur since the machine didn't have enough resources to attend to the number of requests being received simultaneously, so I stopped the test and built the graph with the results Figure 5.15. A total of 3946 requests were attended, with 94.6 % passed and 5.4 % failed requests.

As we can see in the figure above, the configuration that was done caused a saturation point on the machine, where the number of attended requests come down from its peak at 79.20 hits per second to a point of 11 hits per second, and then oscillating as the machine resources are released by the process that attends the request.

With the results, it is concluded that the 750 requests per thread case is less steep to reach the saturation point compared to the 1500 requests per thread case.

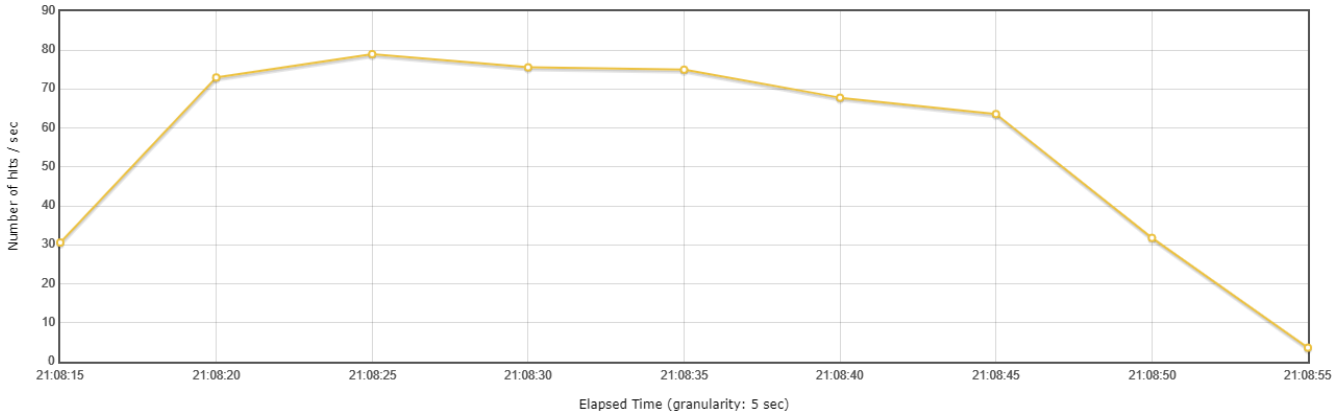

Figure 5.13: Network test 5 threads 500 requests

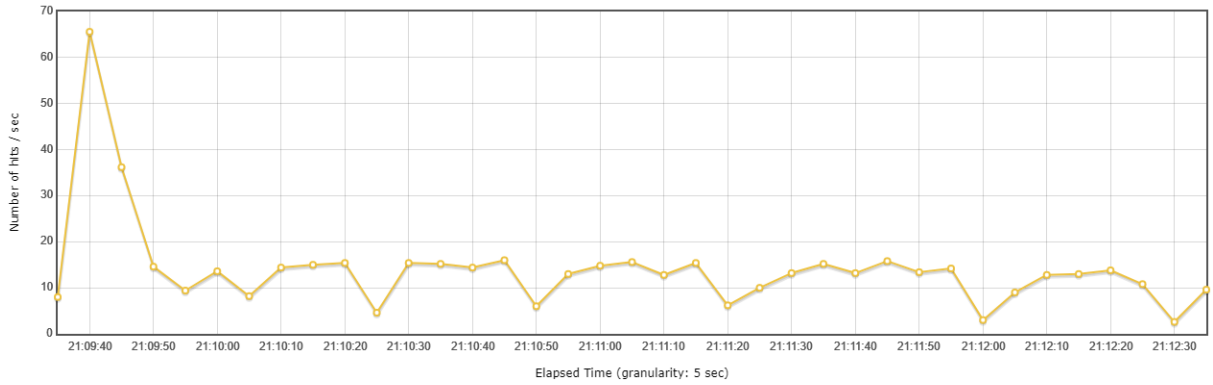

Figure 5.14: Network test 5 threads 750 requests

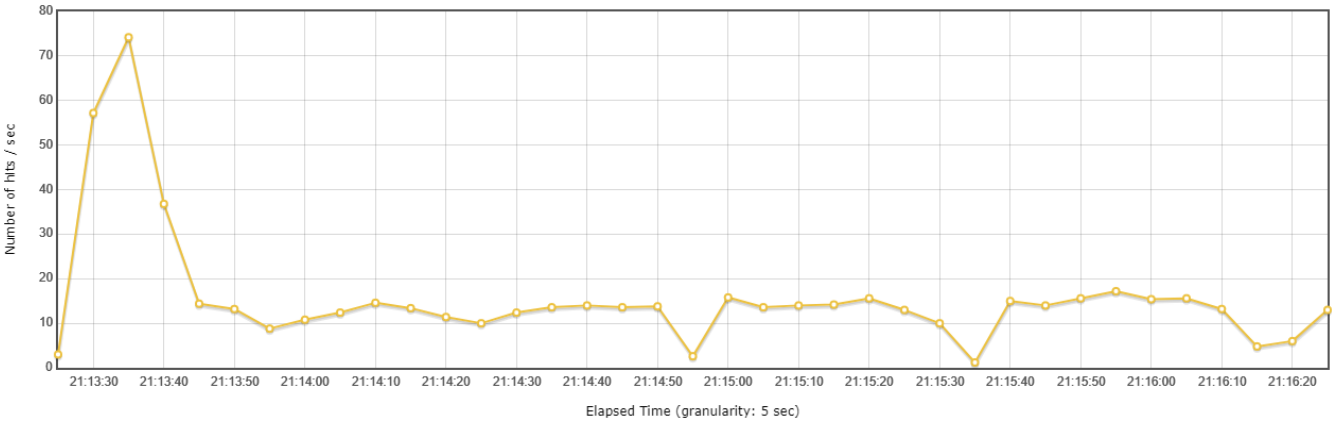

Figure 5.15: Network test 5 threads 1500 requests

5.3 Medium Image Test

This second round of tests resorts to a bigger image, nearly 13 times bigger (228902 bytes image) than the first one in order to compare how the image size influences the number of requests the virtual machine can attend per second. As in the first case, first

	ImageSize (B)	Without Cache (ms)	Cache In Process (ms)	Out of Process Cache (Redis same VM) (ms)	Network Cached (ms)
	228902	50.543697	0.161957	0.646126	25.512394
	228902	45.137873	0.158219	0.78001	25.638556
	228902	46.835951	0.169023	0.575537	25.914919
	228902	46.819836	0.148338	0.644213	25.075125
	228902	45.81241	0.15358	2.808067	25.163552
	228902	46.191168	0.165635	0.814831	25.10867
	228902	44.814667	0.306101	0.650436	24.618648
	228902	46.295527	0.115397	0.628691	25.371256
	228902	49.345101	0.154913	0.774132	25.391621
	228902	46.512497	0.286729	0.531468	24.378216
Average		46.404012	0.160088	0.648281	25.267404
Standard Deviation		2.0156	0.062386	0.057329	0.567089

Table 5.2: 200Kb image processing latencies table

the latency of the requests were tested, with the results in Table 5.2.

Compared to the results in Table 5.1 case, it is safe to say that only the Without cache case has significant differences, in the order of 10 ms. Time to access the caching services were nearly the same between tests with low fluctuations < 2ms.

5.3.1 In-Process stress test

As in the previous case the stress test for the 200Kb image will be also run, in order to conclude how the image size affect the number of requests attended.

On the next test, as it is seen on Figure 5.16 and compared to Figure 5.2 it is evident that the number of attended requests lowers drastically from an average of 16 to 3.5 hits per second. With these results, it can be concluded that the size of the images affects the performance of the functions-framework processing since the resources reserved for the function execution grows as the image size goes up, so to attend one request it will take longer since the processing time is longer and the machine lacks on resources by having them allocated to the function's process.

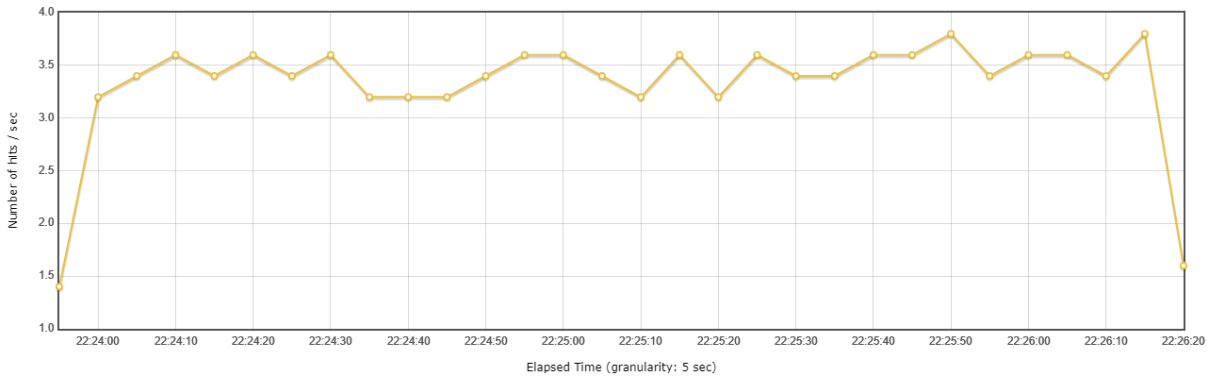

Figure 5.16: In-Process test 1 threads 500 requests

As for the 5 threads, 500 requests on Figure 5.17, and comparing to Figure 5.3 it is also evident a drop in the number of hits per second. The justification also comes from the

memory resources used by the function to process the bigger image. It is also seen that after 1 minute and 50 seconds of the beginning of the test, the number of hits drops to a third of its peak which on the 5 thread, 750 requests test occurs much sooner.

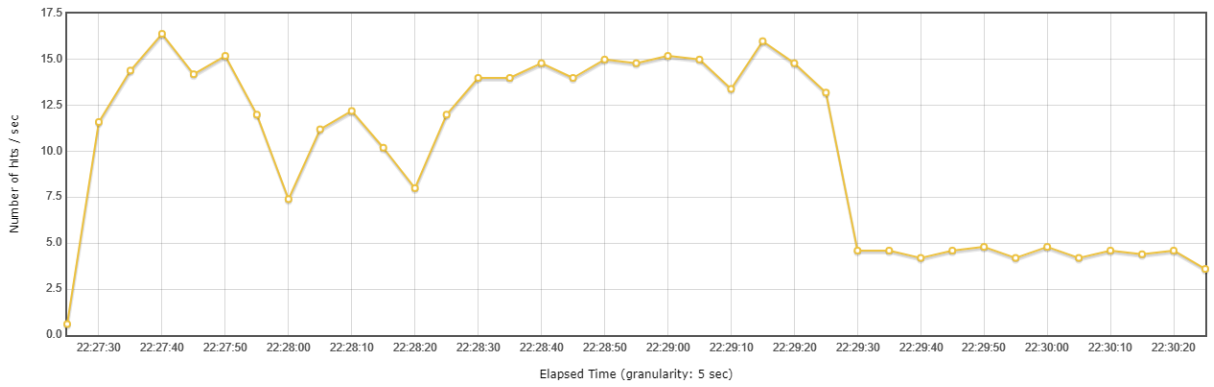

Figure 5.17: In-Process test 5 threads 500 requests

In this test Figure 5.18, and comparing to the previous image, we can see that the saturation point is achieved only after 30 seconds from the beginning of the test. This is justified by the fact that with more requests per thread, the first thread is still making requests when the last one starts, giving less time for the machine to release resources on previous requests.

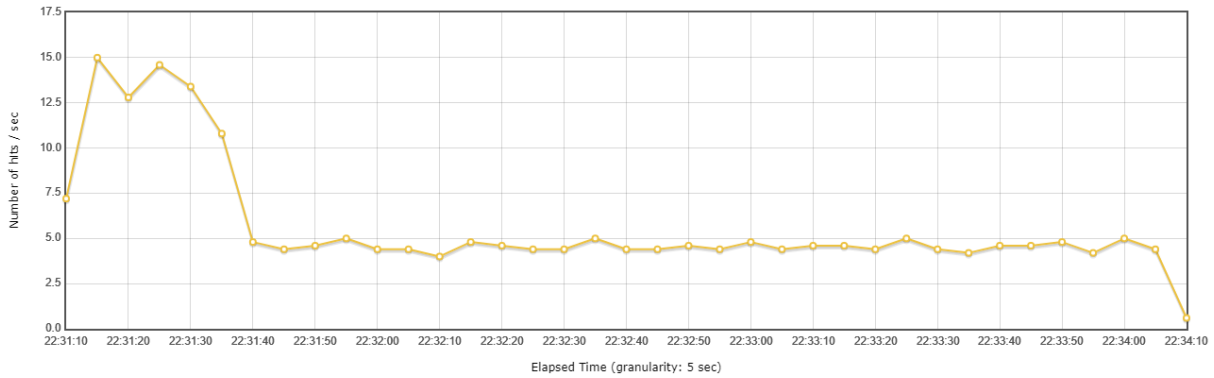

Figure 5.18: In-Process test 5 threads 750 requests

Finally, the same conclusion can be taken from Figure 5.19, where the peak occurs after 15 seconds of the beginning of the test, saturating at 16.80 hits /second coming down to an average of 4.50 hits per second.

5.3.2 Out-of-Process stress test

As for the Out-Of-Process test cases, the goal is the same as the previous set of tests for the lower image. The idea is to understand the differences between hits per second,

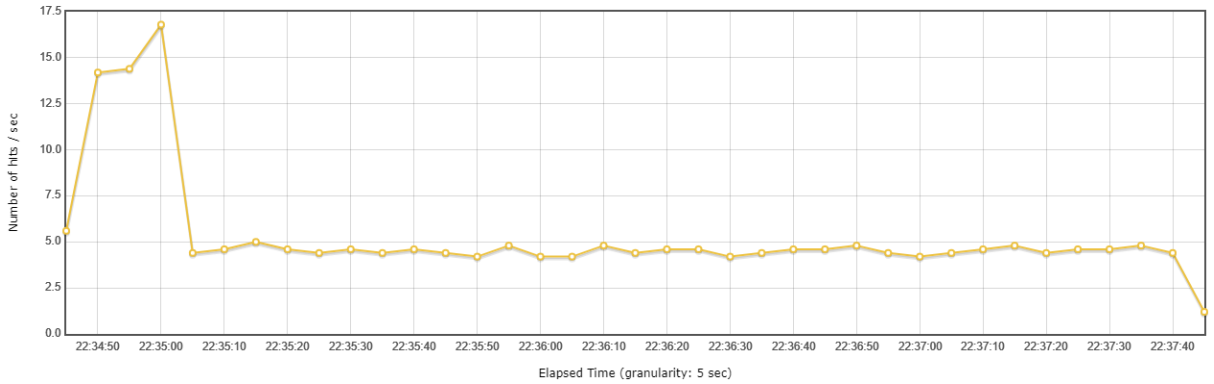

Figure 5.19: In-Process test 5 threads 1500 requests

and the differences on saturation points for each one of the cases. In Figure 5.20 it is again observed an average of 3 hits per second throughout the test.

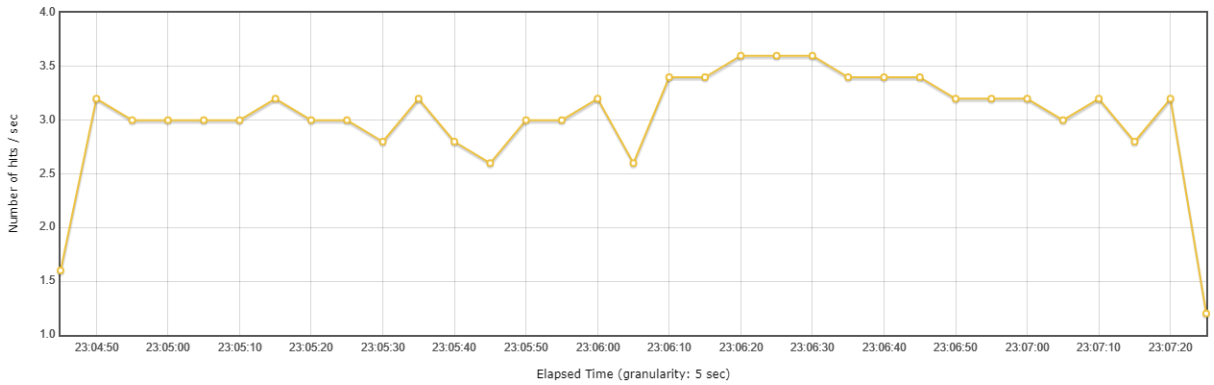

Figure 5.20: Out-Of-Process test 1 threads 500 requests

And as in previous cases when the number of threads goes from 1 to 5, the saturation point appears on the graph Figure 5.21. In this case, saturation occurs after a peak of nearly 15 hits per second and the decay starts after 55 seconds of testing time.

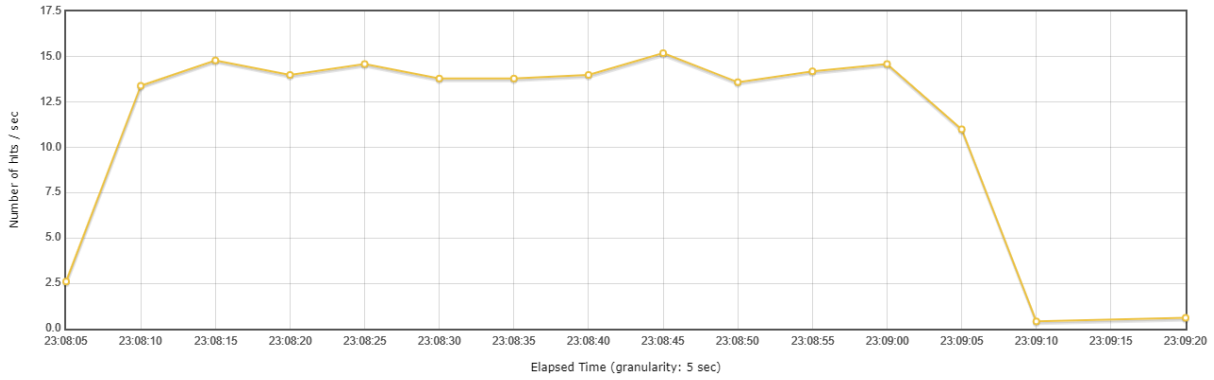

Figure 5.21: Out-Of-Process test 5 threads 500 requests

For Figure 5.22 the peak also occurs after 55 seconds of the test. Compared to the 200Kb

image, In-Process case, we can see that previously the time between saturation points was inferior, and in the out-of-process case, the times are the same comparing the 5 threads 500 requests and the 5 threads 750 requests.

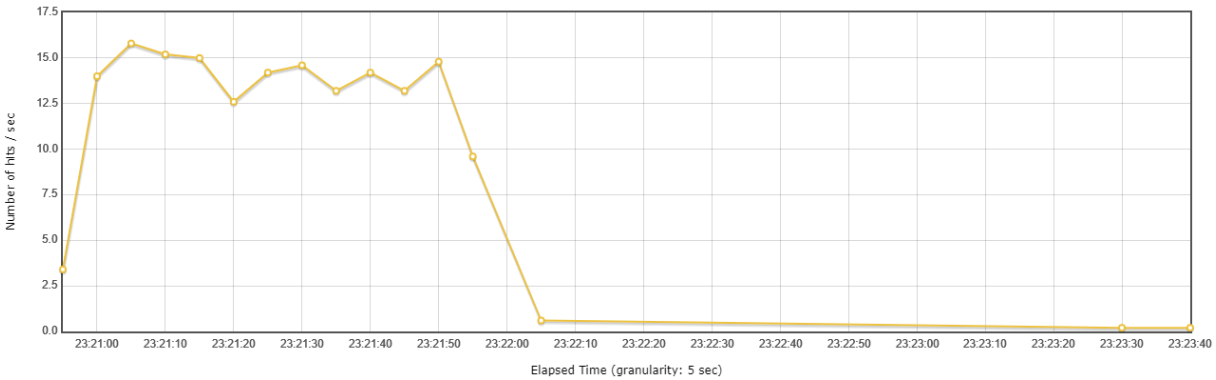

Figure 5.22: Out-Of-Process test 5 threads 750 requests

Finally, for Figure 5.23, the saturation point occurs 1 minute after the test starts, dropping from 14.40 hits per second to 0.60.

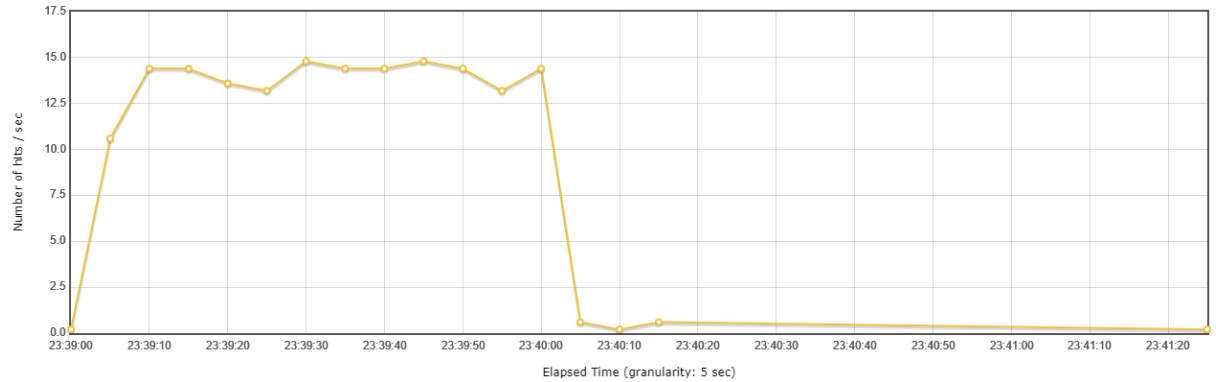

Figure 5.23: Out-Of-Process test 5 threads 1500 requests

5.3.3 Network stress test

For the Network stress test, the average of hits per second as seen in Figure 5.24 is 3.2 with 1 thread and a total of 500 requests.

With 5 threads and 500 requests, the average goes up to 14 hits per second dropping after 1 minute in Figure 5.25 and Figure 5.26 .

Finally with 1500 requests and 5 threads we can see the saturation point occuring after 55 seconds of testing dropping from an average of 16 hits per second to 1.5 Figure 5.27. Comparing to Figure 5.15, it is concluded that the size of the image affects the number of hits per second, going down by 4 to 5 times less than on the previous test. Memory

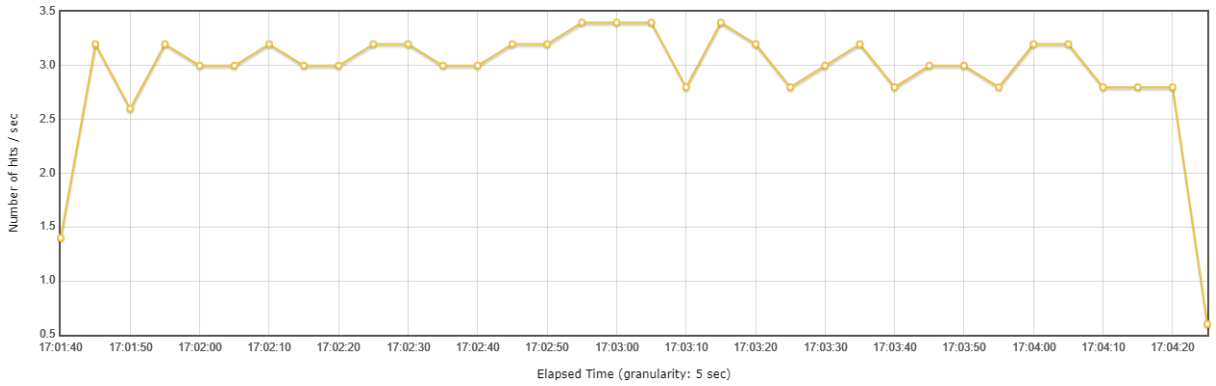

Figure 5.24: Network test 1 thread 500 requests

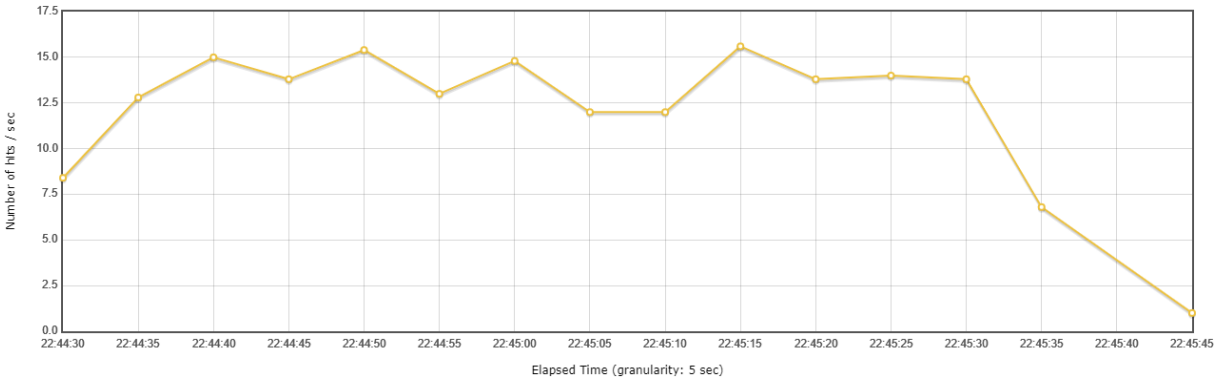

Figure 5.25: Network test 5 threads 500 requests

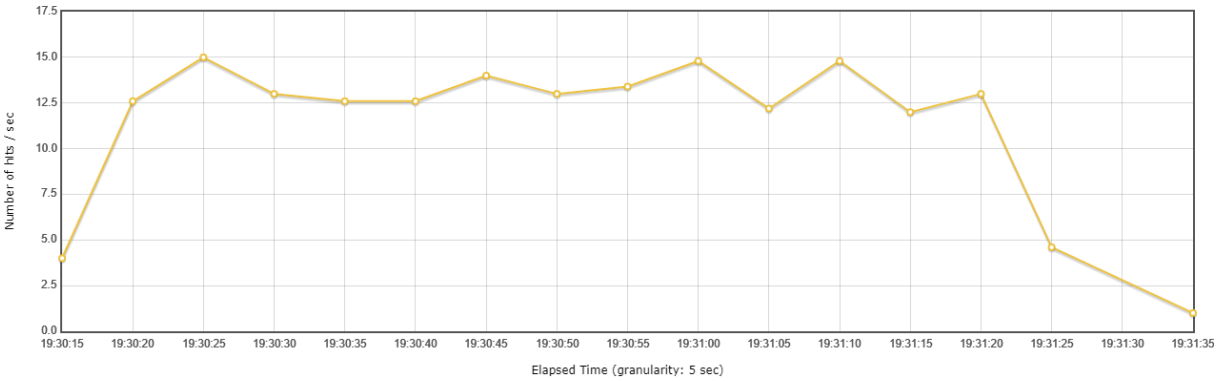

Figure 5.26: Network test 5 threads 750 requests

resources allocated to each function instance are a crucial factor in the performance of the framework, so by using a caching system, the saturation point can be brought further in cases where there are repeated requests where its response can be consulted in the caching system.

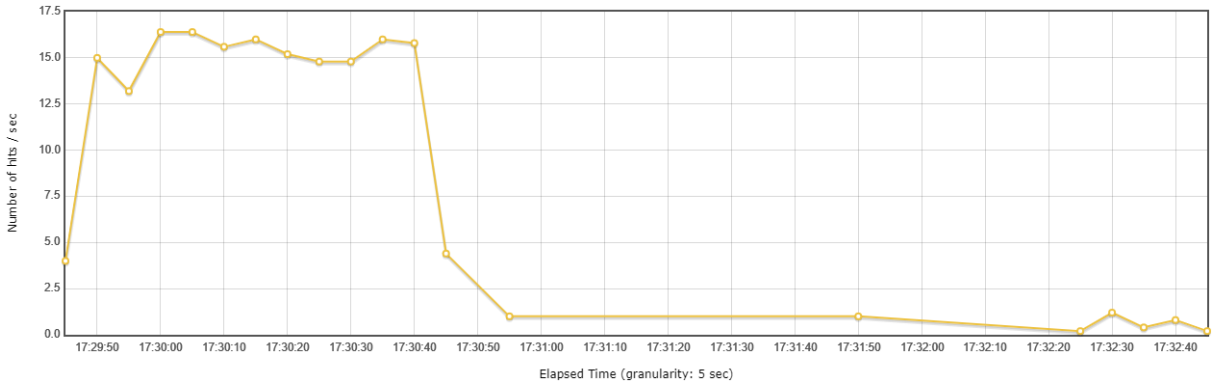

Figure 5.27: Network test 5 threads 1500 requests

	ImageSize (B)	Without Cache (ms)	Cache In Process (ms)	Out of Process Cache (Redis same VM) (ms)	Network Cached (ms)
	406938	352.746978	0.230811	1.45174	24.72248311
	406938	354.811342	0.208635001	1.309533	25.75644411
	406938	328.402607	0.216823	2.722908001	25.81984088
	406938	349.86445	0.246352999	1.770977	25.18952745
	406938	433.565389	0.400401	1.999321001	24.9351189
	406938	362.135653	0.226299001	7.863918001	25.3587
	406938	349.615263	0.381273001	19.008402	24.358658
	406938	319.040788	0.20235	1.616835	25.268256
	406938	316.708354	0.204492001	1.446136	25.12566677
	406938	372.713077	0.191906	1.519962	24.44523578
Average		351.305714	0.221561001	1.693906	25.15759711
Standard Deviation		9.9830495	0.0194525	0.034111	0.138623667

Table 5.3: 4Mb image processing latencies table

5.4 Large Image

As for the large image, the latency values for the processing time in Table 5.3, are higher than the previous cases since the memory and CPU resources needed per request are also larger based on the image size. The saturation point in this case will occur sooner since the number of requests needed to exceed the available resources are little compared to the medium and small images.

For the current test, a 4 Mb image was used as the content of the requests. It is expected that the saturation point will be hit sooner than in previous cases since more resources of the function will be required for its processing.

5.4.1 In-Process stress test

For the In-Process stress test, the average of hits per second as seen in Figure 5.28 is 0.3 with 1 thread and a total of 500 requests. The low amount of hits can be justified by the resources that the function uses to process the image content.

In Figure 5.29 it is seen that the saturation point is reached soon on the function's execution, at a peak of 0.34 hits per second, and going down to less than half the hits per second.

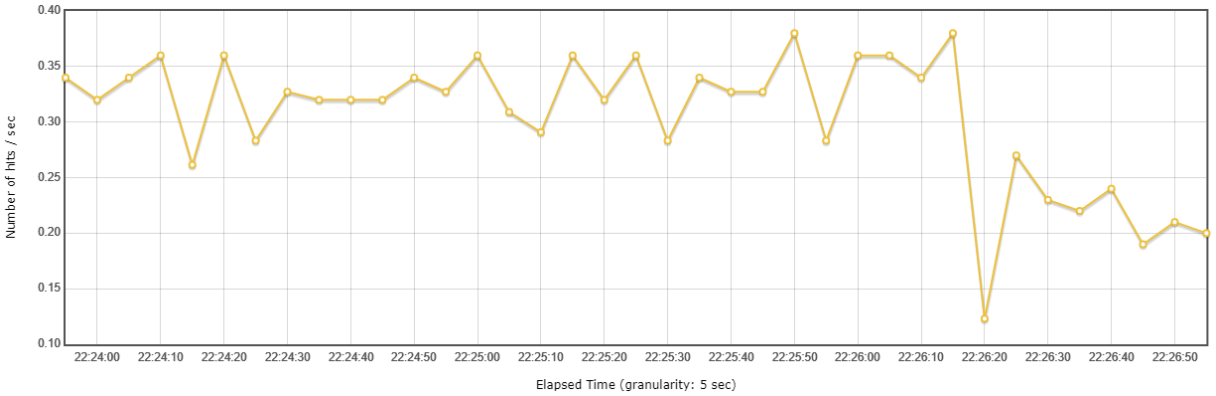

Figure 5.28: InProcess test 1 threads 500 requests

By scaling the number of threads, the load on the framework is higher, and since the image content is heavier than in the previous test cases, the saturation point will be reached much sooner.

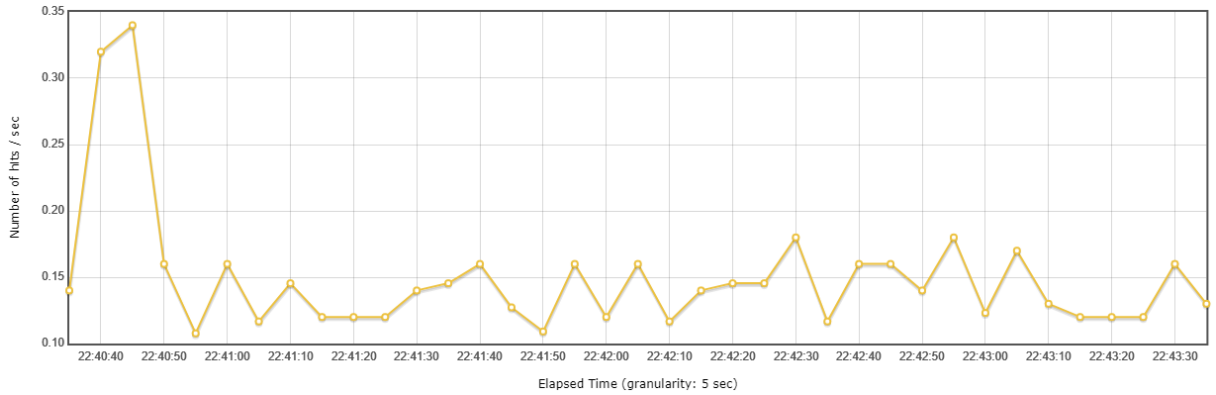

Figure 5.29: InProcess test 5 threads 500 requests

In Figure 5.30 and Figure 5.31 it is seen that the saturation point is reached after 5-10 seconds of execution with a maximum value of 0.35 and 0.3 hits per second being reached.

5.4.2 Out-of-Process stress test

For the Out-of-Process the average number of hits per second goes up when the test configuration has more than 1 thread. In it can be seen that the number of requests per second is on average 0.40, going down at the end of the processing.

In Figure 5.33 it is seen that a peak of hits is reached after 10 seconds, coming to half of the maximum value after this time, and remaining nearly constant during the test time. This means the saturation point was reached, but there were still available resources to attend to requests.

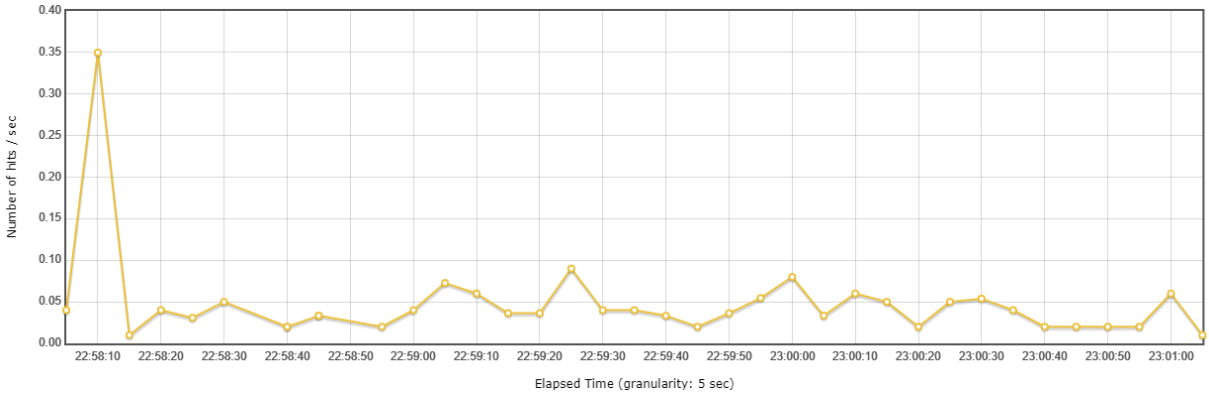

Figure 5.30: Network test 5 threads 750 requests

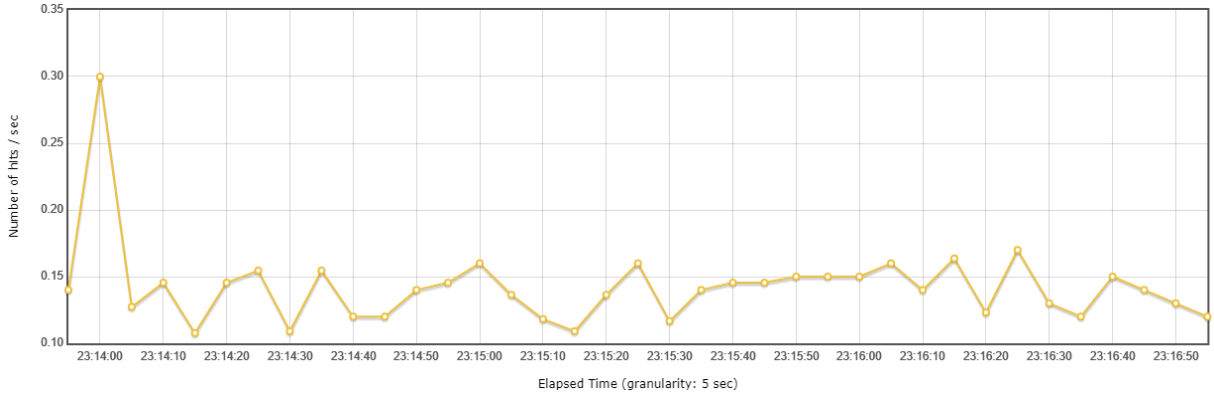

Figure 5.31: Network test 5 threads 1500 requests

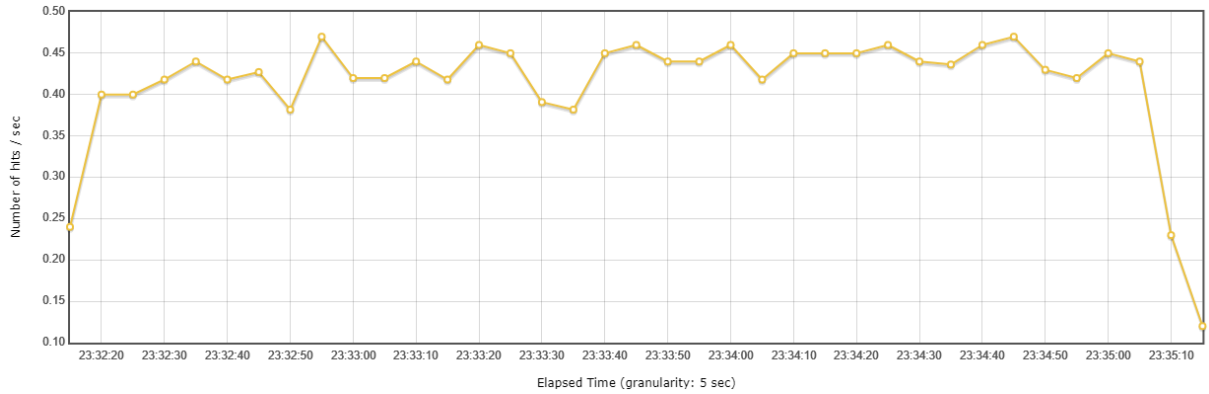

Figure 5.32: Out-Of-Process test 1 threads 500 requests

In Figure 5.34 it is seen that the saturation point is reached after 10 seconds of processing, reaching a maximum 1.27 hits per second and coming down on a ramp shape during the remaining time, until it hits a flat 0 requests per second. This point is reached when there are no resources available, being allocated to the other requests that are waiting for a response from the function.

In Figure 5.35 the saturation point is reached after 10 seconds, crashing to 0 hits per

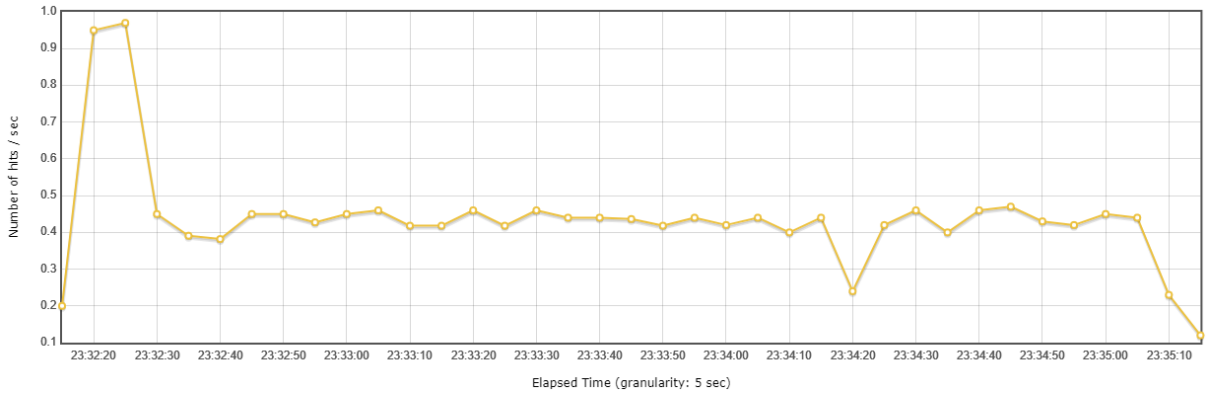

Figure 5.33: Out-Of-Process test 5 threads 500 requests

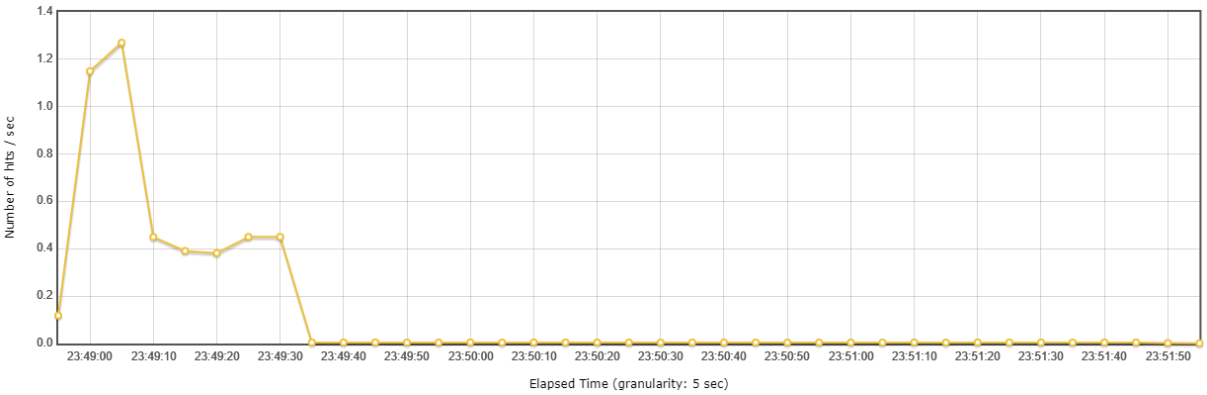

Figure 5.34: Out-Of-Process test 5 threads 750 requests

second. This can be justified by the memory resources being allocated to the requests that were first received by the framework. Since the image is much bigger, the framework takes more time to process the content.

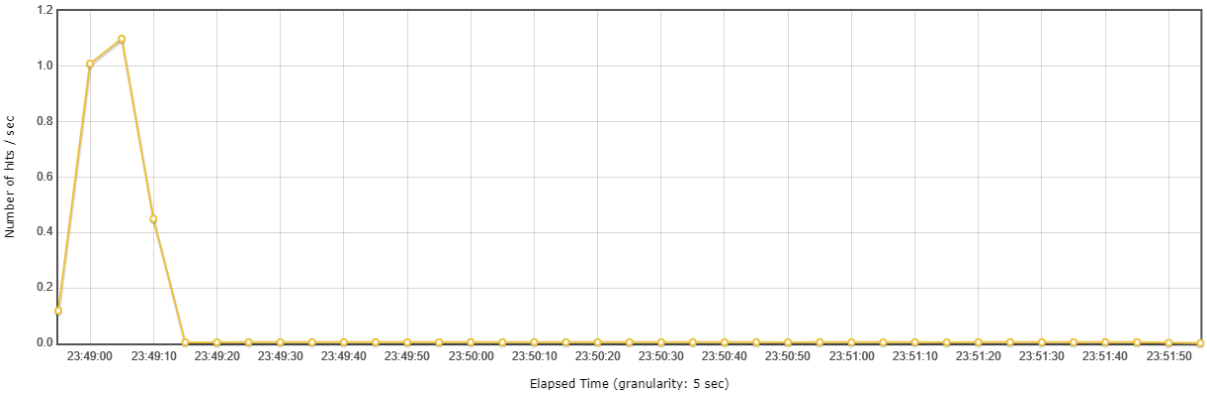

Figure 5.35: Out-Of-Process test 5 threads 1500 requests

5.4.3 Network stress test

For the Network stress test, the average of hits per second as seen in Figure 5.36 is 0.45 with 1 thread and a total of 500 requests.

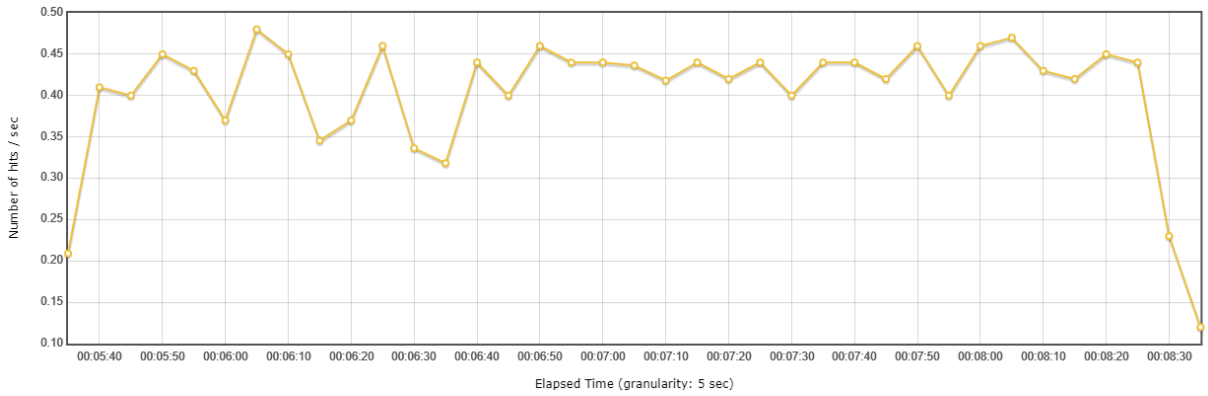

Figure 5.36: Network test 1 thread 500 requests

In Figure 5.37 we can see a ramp figure, that is somewhat similar to Figure 5.34, where the saturation point is reached, but for some seconds some other requests can be attended with a lower frequency, crashing to 0 after 20 seconds.

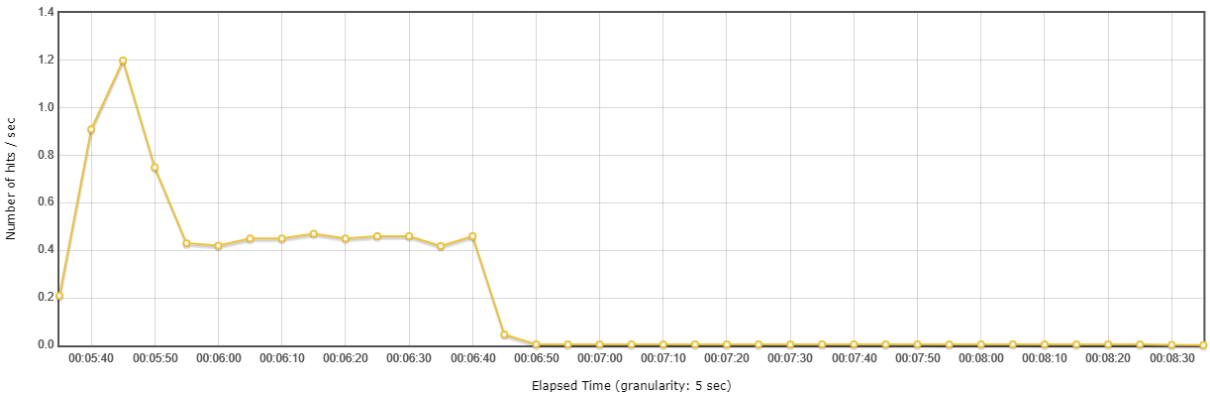

Figure 5.37: Network test 5 threads 500 requests

In Figure 5.38 and Figure 5.39 as in previous cases, it is seen that the saturation point is reached with a maximum of 1.54 and 1.3 hits per second, and coming to 0 after a couple of seconds.

The results obtained on the latest tests, with a large image, show how the limited resources impact the framework's performance to attend requests. Depending on how heavy the processing will be, we can define how the framework will respond, and by using a caching service, these types of problems can be avoided, since the heavy part of the process can be skipped, and the response to a previously processed request can be immediately sent to the client based on the hashed calculated key.

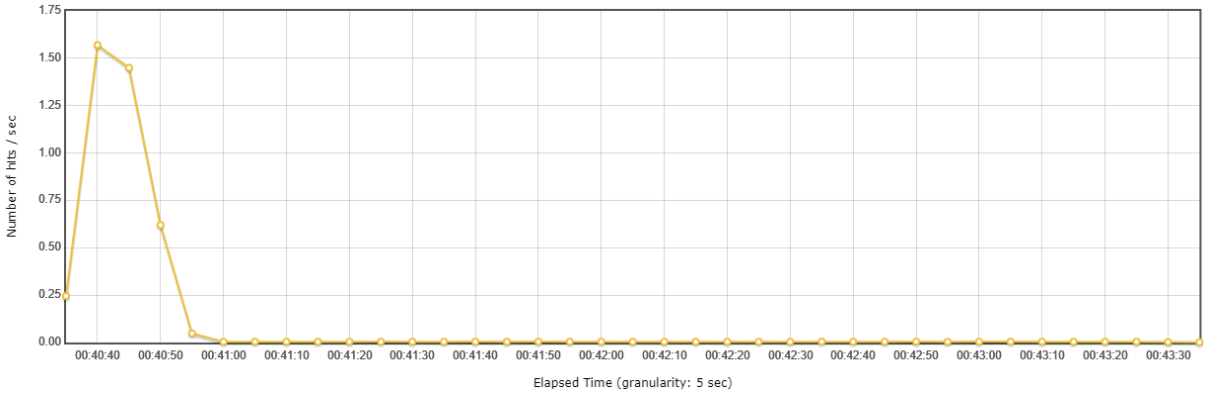

Figure 5.38: Network test 5 threads 750 requests

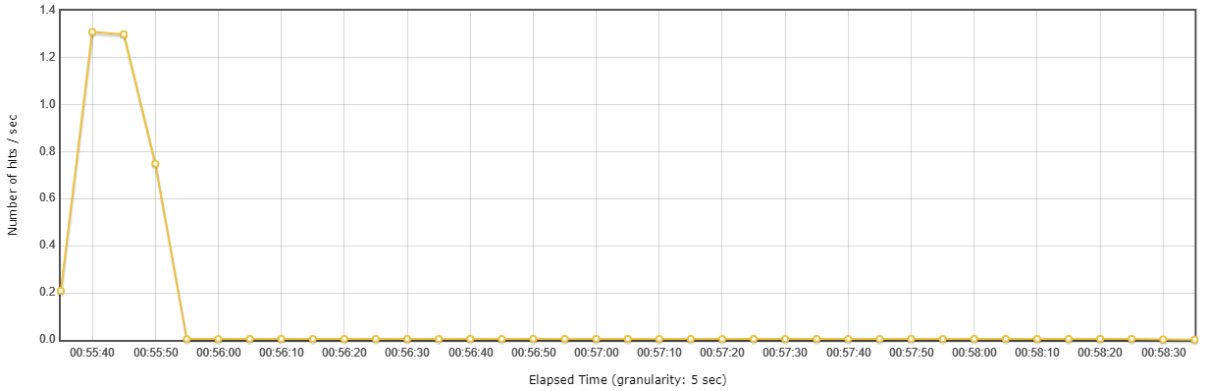

Figure 5.39: Network test 5 threads 1500 requests

Finally, it can be said that the load affects the performance of the framework because the resources are finite, and in order to avoid vertical scaling as a first solution that brings an increment in costs, the caching service can be the mitigation to the problem.

6

Conclusions and Future Work

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 highly elastic environment with the three types of caching systems that have their pros and cons, but show adaptability of the framework to give the end user versatility in its choice.

The evaluation shows that the solution has high performance for images with lower sizes, is able to process a high number of requests on machines with low capacity, providing elasticity if new instances of the function have to be created in different virtual machines, and bringing elasticity to the solution. As for future results it is still being evaluated some testing cases that could bring in in depth data about how the resources of the virtual machine influence the performance of the Functions Framework execution.

As for future work the caching system should be adapted in order to use any type of cache and not be locked to specific software. Also, a very important theme needs to be stated, the caching policy [17], this term refers to the rules that lead to the disposal of cached keys and values since the cache can't grow indefinitely justified by memory limitations inherent in every system [5, 19, 32]. 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 [11].

References

[1] Achilleas Tzenetopoulos, Evangelos Apostolakis, Aphrodite Tzomaka, Christos Papakostopoulos, Konstantinos Stavrakakis, Manolis Katsaragakis, Ioannis Oroutzoglou, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios 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: Springer-Verlag, 2021, 428–438, ISBN: 978-3-030-90538-5. DOI: 10.1007/978-3-030-90539-2_29. [Online]. Available: https://doi.org/10.1007/978-3-030-90539-2_29.

[2] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol", *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, 281–293, 2000, ISSN: 1063-6692. DOI: 10.1109/90.851975. [Online]. Available: https://doi.org/10.1109/90.851975.

[3] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming", *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 5, 890–910, 2004, ISSN: 0164-0925. DOI: 10.1145/1018203.1018208. [Online]. Available: https://doi.org/10.1145/1018203.1018208.

[4] S. Masoud Sadjadi, Philip K. McKinley, and Betty 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*, ser. DEAS '05, St. Louis, Missouri: Association for Computing Machinery, 2005, 1–7, ISBN: 1595930396. DOI:

10 . 1145 / 1083063 . 1083086. [Online]. Available: https://doi.org/10.1145/1083063.1083086.

[5] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer, "Adaptive insertion policies for high performance caching", *SIGARCH Comput. Archit. News*, vol. 35, no. 2, 381–391, 2007, ISSN: 0163-5964. DOI: 10 . 1145 / 1273440 . 1250709. [Online]. Available: https://doi.org/10.1145/1273440.1250709.

[6] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici, "My vm is lighter (and safer) than your container", in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, 218–233, ISBN: 9781450350853. DOI: 10 . 1145 / 3132747 . 3132763. [Online]. Available: https://doi.org/10.1145/3132747.3132763.

[7] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter, "Sprocket: A serverless video processing framework", in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, Carlsbad, CA, USA: Association for Computing Machinery, 2018, 263–274, ISBN: 9781450360111. DOI: 10.1145/3267809.3267815. [Online]. Available: https://doi.org/10.1145/3267809.3267815.

[8] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo, "Seuss: Skip redundant paths to make serverless fast", in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10 . 1145 / 3342195 . 3392698. [Online]. Available: https://doi.org/10.1145/3342195.3392698.

[9] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach, "Usetl: Unikernels for serverless extract transform and load why should you settle for less?", in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19, Hangzhou, China: Association for Computing Machinery, 2019, 23–30, ISBN: 9781450368933. DOI: 10 . 1145 / 3343737 . 3343750. [Online]. Available: https://doi.org/10.1145/3343737.3343750.

[10] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Christian 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*, ser. UCC'19, Auckland, New Zealand: Association for Computing Machinery, 2019, 273–283,

ISBN: 9781450368940. DOI: 10.1145/3344341.3368813. [Online]. Available: https://doi.org/10.1145/3344341.3368813.

[11] John T. Robinson and Murthy V. Devarakonda, "Data cache management using frequency-based replacement", *SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, 134–142, 1990, ISSN: 0163-5999. DOI: 10.1145/98460.98523. [Online]. Available: https://doi.org/10.1145/98460.98523.

[12] Justice Opara-Martins, Reza Sahandi, and Feng Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective", *J. Cloud Comput.*, vol. 5, no. 1, 2016, ISSN: 2192-113X. DOI: 10.1186/s13677-016-0054-z. [Online]. Available: https://doi.org/10.1186/s13677-016-0054-z.

[13] Bradley M. Duska, David Marwood, and Michael J. Feeley, "The measured access characteristics of world-wide-web client proxy caches", in *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, ser. USITS'97, Monterey, California: USENIX Association, 1997, page 3.

[14] Pei Cao and Sandy Irani, "Cost-aware www proxy caching algorithms", in *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, ser. USITS'97, Monterey, California: USENIX Association, 1997, page 18.

[15] Steven D. Gribble and Eric A. Brewer, "System design issues for internet middleware services: Deductions from a large client trace", in *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, ser. USITS'97, Monterey, California: USENIX Association, 1997, page 19.

[16] Peter M. Mell and Timothy Grance, "Sp 800-145. the nist definition of cloud computing", Gaithersburg, MD, USA, Tech. Rep., 2011.

[17] 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.*, 2004, pages 182–189. DOI: 10.1109/CCGrid.2004.1336565.

[18] J. Korhonen and Y. Wang, "Effect of packet size on loss rate and delay in wireless links", in *IEEE Wireless Communications and Networking Conference, 2005*, vol. 3, 2005, 1608–1613 Vol. 3. DOI: 10.1109/WCNC.2005.1424754.

[19] M.K. Qureshi, D. Thompson, and Y.N. Patt, "The v-way cache: Demand-based associativity via global replacement", in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pages 544–555. DOI: 10.1109/ISCA.2005.52.

[20] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein, "Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads", in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pages 363–376, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[21] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt, "SAND: Towards High-Performance serverless computing", in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pages 923–935, ISBN: 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/akkus.

[22] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau, "SOCK: Rapid task provisioning with Serverless-Optimized containers", in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pages 57–70, ISBN: 978-1-931971-44-7. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oakes.

[23] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift, "Peeking behind the curtains of serverless platforms", in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pages 133–146, ISBN: ISBN 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang.

[24] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics", in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018,

pages 427–444, ISBN: 978-1-939133-08-3. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[25] Qifan Pu, Shivaram Venkataraman, and Ion Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure", in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pages 193–206, ISBN: 978-1-931971-49-2. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu.

[26] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov, "Agile cold starts for scalable serverless", in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/mohan.

[27] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue 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: USENIX Association, Feb. 2020, pages 267–281, ISBN: 978-1-939133-12-0. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/wang-ao.

[28] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa, "Firecracker: Lightweight virtualization for serverless applications", in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pages 419–434, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache.

[29] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider", in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pages 205–218, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad.

[30] Simon Shillaker and Peter Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing", in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pages 419–433, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker.

[31] Mikio Kataoka, Kunihiko Toumura, Hideki Okita, Junji Yamamoto, and Toshiaki Suzuki, "Distributed cache system for large-scale networks", in *2006 International Multi-Conference on Computing in the Global Information Technology - (ICCGI'06)*, 2006, pages 40–40. DOI: 10.1109/ICCGI.2006.26.

[32] T.L. Johnson, D.A. Connors, and W.W. Hwu, "Run-time adaptive cache management", in *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, vol. 7, 1998, 774–775 vol.7. DOI: 10.1109/HICSS.1998.649283.

[33] Rigzin Angmo and Monika Sharma, "Performance evaluation of web based automation testing tools", in *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, 2014, pages 731–735. DOI: 10.1109/CONFLUENCE.2014.6949287.

[34] Jinglu Zhang and Zhao Yan, "Middleware technology research and interface design in ubiquitous network", in *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*, 2015, pages 659–662. DOI: 10.1109/ICMTMA.2015.164.

[35] Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao, and Minyi Guo, "Towards scalable and reliable in-memory storage system: A case study with redis", in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pages 1660–1667. DOI: 10.1109/TrustCom.2016.0255.

[36] Rabiya Abbas, Zainab Sultan, and Shahid Nazir Bhatti, "Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege", in *2017 International Conference on Communication Technologies (ComTech)*, 2017, pages 39–44. DOI: 10.1109/COMTECH.2017.8065747.

[37] Sunil Kumar Mohanty, Gopika Premsankar, and Mario di Francesco, "An evaluation of open source serverless computing frameworks", in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, pages 115–120. DOI: 10.1109/CloudCom2018.2018.00033.

[38] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz, "Cold start influencing factors in function as a service", in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pages 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[39] Mikhail M. Rovnyagin, Valentin K. Kozlov, Roman A. Mitenkov, Alexey D. Gukov, and Anton 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)*, 2020, pages 468–471. DOI: 10.1109/EIConRus49466.2020.9039502.

[40] Nitin Sukhija and Elizabeth Bautista, "Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus", in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2019, pages 257–262. DOI: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.

[41] Tobias Pfandzelter and David Bermbach, "Tinyfaas: A lightweight faas platform for edge environments", in *2020 IEEE International Conference on Fog Computing (ICFC)*, 2020, pages 17–24. DOI: 10.1109/ICFC49376.2020.00011.

[42] David Balla, Markosz Maliosz, and Csaba Simon, "Open source faas performance aspects", in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, 2020, pages 358–364. DOI: 10.1109/TSP49548.2020.9163456.

[43] Ruchika Muddinagiri, Shubham Ambavane, and Simran Bayas, "Self-hosted kubernetes: Deploying docker containers locally with minikube", in *2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, 2019, pages 239–243. DOI: 10.1109/ICITAET47105.2019.9170208.

[44] Xiaoping Huang, "Research and application of node.js core technology", in *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, 2020, pages 1–4. DOI: 10.1109/ICHCI51889.2020.00008.

[45] Mohammad Alhowaidi, Deepak Nadig, Boyang Hu, Byrav Ramamurthy, and Brian Bockelman, "Cache management for large data transfers and multipath forwarding strategies in named data networking", *Computer Networks*, vol. 199, page 108 437, 2021, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.c

omnet.2021.108437. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128621003972.

[46] *Apache JMeter - Apache JMeter™*. [Online]. Available: https://jmeter.apache.org/.

[47] Dariusz R. Augustyn, Łukasz Wyciślik, and Mateusz Sojka, "The faas-based cloud agnostic architecture of medical servicesmdash;polish case study", *Applied Sciences*, vol. 12, no. 15, 2022, ISSN: 2076-3417. DOI: 10.3390/app12157954. [Online]. Available: https://www.mdpi.com/2076-3417/12/15/7954.

[48] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz, "Cirrus: A serverless framework for end-to-end ml workflows", *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[49] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu, "Serverless computing: One step forward, two steps back", *CoRR*, vol. abs/1812.03651, 2018. arXiv: 1812.03651. [Online]. Available: http://arxiv.org/abs/1812.03651.

[50] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek, "Kubernetes as an availability manager for microservice applications", *CoRR*, vol. abs/1901.04946, 2019. arXiv: 1901.04946. [Online]. Available: http://arxiv.org/abs/1901.04946.

[51] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li, "Understanding open source serverless platforms: Design considerations and performance", *CoRR*, vol. abs/1911.07449, 2019. arXiv: 1911.07449. [Online]. Available: http://arxiv.org/abs/1911.07449.

[52] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov, "Cloudburst: Stateful functions-as-a-service", *CoRR*, vol. abs/2001.04592, 2020. arXiv: 2001.04592. [Online]. Available: https://arxiv.org/abs/2001.04592.

[53] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications", *CoRR*, vol. abs/2104.13869, 2021. arXiv: 2104.13869. [Online]. Available: https://arxiv.org/abs/2104.13869.

[54] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute", *CoRR*, vol. abs/2105.11229, 2021. arXiv: 2105.11229. [Online]. Available: https://arxiv.org/abs/2105.11229.

[55] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter, "Serverless computing: Current trends and open problems", *CoRR*, vol. abs/1706.03178, 2017. arXiv: 1706.03178. [Online]. Available: http://arxiv.org/abs/1706.03178.

[56] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting", *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[57] Njeru Mwendi Edwin, "Software frameworks, architectural and design patterns", *Journal of Software Engineering and Applications*, vol. 07, no. 08, pages 670–678, 2014. DOI: 10.4236/jsea.2014.78061. [Online]. Available: https://doi.org/10.4236/jsea.2014.78061.

[58] *Pub/sub: A google-scale messaging service | google cloud*, 2022. [Online]. Available: https://cloud.google.com/pubsub/architecture.

[59] given=GoogleCloudPlatform, *GitHub - GoogleCloudPlatform/functions-framework-nodejs: FaaS (Function as a service) framework for writing portable Node.js functions*. [Online]. Available: https://github.com/GoogleCloudPlatform/functions-framework-nodejs.

[60] Kim Long Ngo, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu, *Has your faas application been decommissioned yet? – a case study on the idle timeout in function as a service infrastructure*, 2022. DOI: 10.48550/ARXIV.2203.10227. [Online]. Available: https://arxiv.org/abs/2203.10227.

[61] Pablo Gimeno Sarroca and Marc Sánchez-Artigas, *Mlless: Achieving cost efficiency in serverless machine learning training*, 2022. DOI: 10.48550/ARXIV.2206.05786. [Online]. Available: https://arxiv.org/abs/2206.05786.

[62] Rizwan Khan, "Comparative study of performance testing tools: Apache jmeter and hp loadrunner", 2016.

[63] *Kubernetes documentation*. [Online]. Available: https://kubernetes.io/docs/home/.

[64] Mathias Peter, *npm: node-cache*, Jul. 2020. [Online]. Available: https://www.npmjs.com/package/node-cache.

[65] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël de Palma, Bernabé Batchakui, and Alain Tchana, "OFC: an opportunistic caching system for FaaS platforms", in *EUROSYS 2021 - 16th European Conference on Computer Systems*, Edinburgh (online), United Kingdom: Association for Computing Machinery (ACM), Apr. 2021, pages 228–244. DOI: 10.1145/3447786.3456239. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03211416.

[66] Node-Cache, *Node-cache/node-cache: A node internal (in-memory) caching module*. [Online]. Available: https://github.com/node-cache/node-cache.

[67] *MS Windows NT kernel description*, https://ericstoekl.github.io/faas/architecture/, Accessed: 2021-11-16.

[68] Danilo Pochia, *New for aws lambda – container image support*, 2020. [Online]. Available: https://aws.amazon.com/pt/blogs/aws/new-for-aws-lambda-container-image-support/.

[69] [Online]. Available: https://redis.io/.

[70] Ivan Rosa, *EfficientFaaS*, version 1.0.0, Dec. 2022. DOI: 10.5281/zenodo.1234. [Online]. Available: https://github.com/ivansocket/EfficientFaaS.

[71] *npm: image-thumbnail*, May 2022. [Online]. Available: https://www.npmjs.com/package/image-thumbnail.

[72] *Crypto | Node.js v19.2.0 Documentation*. [Online]. Available: https://nodejs.org/api/crypto.html.

[73] *General-purpose machine family | Compute Engine Documentation |*. [Online]. Available: https://cloud.google.com/compute/docs/general-purpose-machines.

[74] Jorge Villalon and Rafael Calvo, *A decoupled architecture for scalability in text mining applications*. 2013. DOI: 10.3217/jucs-019-03-0406.

[75] Wikipedia, *SHA-2 — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=SHA-2&oldid=1124499017, [Online; accessed 15-February-2022], 2022.

[76] James Wirth, *Anatomy of a functions as a service (faas) solution*, 2020. [Online]. Available: https://blogs.vmware.com/customer-experience-and-success/2020/04/anatomy-of-a-functions-as-a-service-faas-solution.html.

[77] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Christian 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)*, ACM, Dec. 2019, pages 273–283. DOI: 10.1145/3344341.3368813.