# OPTIMIZING COLD START LATENCY IN SERVERLESS COMPUTING

A Thesis

by

NIKHIL PREMANAND BHAT

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2020

Major Subject: Computer Science

## ABSTRACT

Serverless computing has gained traction in public cloud offerings, including AWS, Azure, and GCP, in the past few years. Consumers of these platforms cherish the ability to write multiple functions without much need to write boilerplate code or to manage these servers. However, despite its benefits, cloud computing suffers high latency when reacting to intermittent events, due to the cost of deploying function code and data to new instances and the cost of initializing the sandboxed function runtime–known as the cold start latency. Cold start latency poses a major challenge to the burst-out performance of serverless applications. In this thesis, we analyze the components of cold start latency in commercial serverless platforms to understand the factors of networking and the impact of function binary sizes. We further propose a solution to build a framework to reduce the latency of data transfer when initializing a function on a new instance. We use the idea of deduplication and data encoding, to reuse data blocks from a global corpus of frequently used blocks, or to reuse blocks that previously occur on the same machines. We build a framework for general optimization of data movement between cloud servers and demonstrate that with a high data reuse rate, the network component of the cold start latency can be significantly reduced.

## DEDICATION

To my mother and my father, who have been a constant source of support and inspiration for all my endeavours in life.

## ACKNOWLEDGMENTS

I would like to express my deepest appreciation and gratitude to Prof. Tsai for his guidance and mentorship in carrying out the research work detailed in this thesis. I would also like to acknowledge the contributions of Akhila Mangipudi and Abhishek Singh, who helped me with certain aspects of experimentation and design as part of the research.

I would also like to thank my fellow lab partners, Manvitha Reddy and Gautham Srinivasan, as well as my friends, Prakhar Mohan, Vansh Narula, Manish Patel and Abhishek Das for their constant support and encouragement, especially during my late evening endeavours.

## CONTRIBUTORS AND FUNDING SOURCES

The thesis committee for this work include Dr. Chia-Che Tsai (Chair) from the Department of Computer Science, Dr. Dilma Da Silva of the Department of Computer Science and Dr. Narasimha Reddy of the Department of Electrical Engineering at Texas A& M University.

The design for a global encoding scheme, presented in section 6.3.2 was developed in collaboration with Akhila Mangipudi. Akhila developed the idea to use huffman encoding scheme for shorter keys for global chunks, and designed a prototype to demonstrate the effectiveness of such an implementation.

The experiments carried out to find an effective chunking strategy, presented in section 5.2, were carried out in collaboration with Abhishek Singh.

The work was funded by Dean Start-Up Grant 02-241570 from Texas A&M University.

NOMENCLATURE

| | |
|---|---|
| AWS | Amazon Web Services |
| HTTP | Hypertext Transfer Protocol |
| EC2 | Elastic Cloud Compute |
| S3 | Simple Storage Service |
| SQS | Simple Queue Service |
| SQS | Simple Notification Service |
| FaaS | Function as a Service |
| BaaS | Backend as a Service |
| API | Application Program Interface |
| CLI | Command Line Interface |
| GRPC | Google Remote Proceddure Call |
| JDK | Java Development Kit |

TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1. INTRODUCTION

## 1.1 Serverless Computing

Serverless computing is a new paradigm in the cloud computing industry. It provides an execution model where the cloud vendors provide interfaces to the customers to write their code as a 'function' in a high-level language. The cloud vendors wrap the function code into a suitable web application (typically a web server) and managing its deployment and scaling. The consumer of the platform is completely agnostic to the deployment and administration of the application.

Serverless functions are constrained by specific architectural properties by which they differ from a traditional computing model. To begin with, functions are event-based. The customer can configure the type of event from a wide range of options such as an HTTP request, addition of an entry in a message queue, etc. Another characteristic feature of functions is that they are short-lived. The life of a function can be configured up to a limit of 15 minutes typically. These two properties of a function enable serverless computing to be provided to the customers on an 'on-demand' basis and a pay-per-use business model. The customer can pay the cloud vendor only in proportion to the resources used rather than a fixed amount of resources allocated.

While the functionality provided by the various serverless platforms have their minor differences, their design is majorly guided by the same set of architectural principles: Stateless design, Automatic scaling, Limited maximum execution time, Event-based triggering, and wide support for major high-level languages such as Node.js, Python, .NET, etc.

### 1.1.1 Evolution of Serverless Computing

Serverless platforms can be best understood by inspecting the evolution of the cloud computing model. Figure 1.1 describes a simple chart explaining this evolution. Before the advent of cloud computing in around 2008, the traditional computing model consisted of developers building web applications and handing it over to the operations personnel, who would then deploy them on physical hardware servers either owned and located on-premises or maintained in co-location centers.

Release cycles would generally be long, and deploying a globally distributed robust and reliant network would take mammoth efforts.

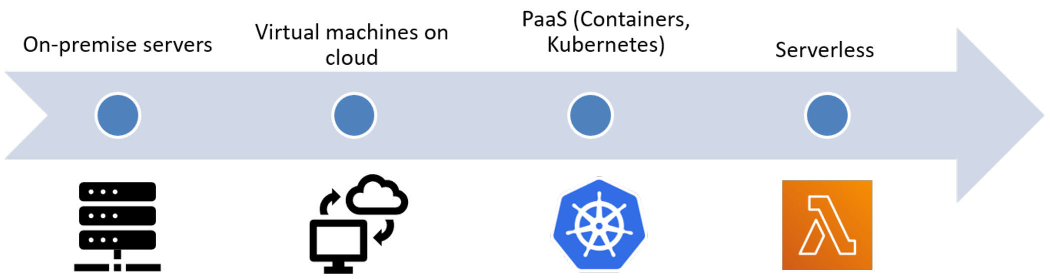


Figure 1.1: Evolution of Serverless Computing

The development of a shared public computing model, pioneered by Google's App Engine and Amazon's EC2, [8] brought a revolution in the way web applications were deployed. These developments were also supplemented by significant advancements made in the building of large scale warehouse-level data centers, virtualization, and networking. [9] Cloud computing heavily simplified the process of development and deployment of web applications. Efforts to buy and maintain physical servers were no longer required for the customers and sharing of resources among multiple customers enabled higher utilization and efficient of resources. Scaling of resources up and down became easy. Customers largely benefited from paying only for the number of virtual servers they owned at any given time.

Since the advent of cloud computing, significant advancements were made to further ease the process of development and deployment of web applications. Cloud vendors started offering a huge variety of services to be included in the architectures of their web applications. This fueled a paradigm shift in the way web based software is designed from traditionally monolithic designs to microservice based architecture. Customers also started using services readily available of the cloud such as blob storages, key value-stores, message queues, etc. Another important innovation which fueled the wider adoption of microservices based architecture was the advancements made in lightweight virtualization techniques such as containerization. Developers could now package their microservices in containers which helped in making sure that the containers could be directly deployed on the infrastructure without much of variability introduced by the production environment. The convenience of having a truly ’build once, run anywhere’ paradigm helped in greatly reducing the efforts of operation and maintenance.

Containerization was complemented by the subsequent development of containerization orchestrations services such as Kubernetes and Docker swarm. These services enabled deployment of containers as services in a distributed computing environment, and also provide a wide range of options to dynamically scale the services, distribute workload with load balancing etc. A large amount of maintenance operations could be replaced with automated policies. Continuous Integration and Delivery pipelines could now be setup to directly deploy updates to the existing infrastructure by the developers, often with limited to no support from Operations teams.

| **Functionality** | **Virtual Machines on Cloud** | **PaaS offerings** | **Serverless Computing** |
|---|---|---|---|
| Language support | Any | Any | Wide but limited( C#, Java, Node.js, Go) |
| Program run | Always running | Always running but can be scaled on demand | Event based trigger |
| Scaling | Manual | Automated via policies | Automated and completely abstracted from customer |
| Maximum Run time | None | None | 90 seconds |
| Maximum Memory allowed | None | None | 256-512 MB |
| Permanent storage | Internal/ External | External | External |
| Application Framework | Any | Any | Provided by cloud vendor |
| Operating System | Any | Any | Provided by cloud vendor |

Table 1.1: Comparison of cloud offerings to build web applications

Continuing this pattern of abstraction one step further, Amazon in 2014 introduced the Amazon Lambda, which gave rise to the paradigm of 'Function as a service'(FaaS). Instead of giving developers the freedom to use their own frameworks and configuration for building their web application, the serverless platform abstracts them. The developers just have the ability to write their business logic inside designated sections of code templates called function handlers and the platform takes care of the job of making a web application out of this, as well as managing its deployment and scaling. Table 1.1 describes the fundamental differences between the architecture of serverless computing platforms and other cloud offerings.

### 1.1.2 Backend as a Service

Functions in isolation can be used to achieve only limited amount of functionalities. It is often the integration of functions in a larger architecture with supplementing services which has contributed to the success and utility of functions at such a large scale.

One such important service is the availability of object storage services, such as the Amazon Simple Storage Service(s3).[10] A function is short lived and hence cannot store state information inside its memory or local disk. Since their life is limited, they can get terminated at any time after a particular event is served. This necessitates the need of an external storage service to store state information. While many a times it can be done via provisioning dedicated databases, customers are often looking for simplicity while building functions. When in need of a simple service with an easy interface to maintain state, it is often achieved using services like s3.

Another such important backend service is a message queue, which is often used to communicate between the various components of the backend architecture. These message queues can either be push based such as the Amazon Simple Notification Service (SNS)[11] or poll based such as Amazon Simple Queue Service (SQS)[12]. Messages in the queue can be used as events to trigger the functions.

Hence, while functions are placed at the core of the Serverless computing domain, services such as AWS S3 and SNS fall under the paradigm of Backend as a service. There are plenty other services which fall under these category, which can be pulled out of the box to achieve functionalities such as authentication, social media integration, etc. Serverless computing is often regarded as the composition of FaaS and BaaS. [8]

### 1.1.3 Serverless Platforms

There are plenty of platforms available today which provide the benefits of Serverless computing. AWS Lambda [13] was the initial offering in this domain, introduced in 2015. This was soon followed by Microsoft releasing Azure functions [14] and Google releasing Google functions [15].

Apart from these vendor-specific serverless platforms, there are also some open-source alternatives that are gaining traction and on the rise. OpenWhisk [16] and OpenFaas [17] are the most popular platforms in this category.

Certain application-level frameworks such as Serverless frameworks have been built to provide an abstraction over the popular serverless platforms. They provide their own interfaces to write functions, which are then translated to the vendor-specific function interfaces. This enables the customers to writing functions that can be run on any of these platforms with minimum changes.

### 1.1.4 Function event triggers

As discussed in 1.1 , functions are triggered based on events. While most common type of an event maybe a http request, there is a wide range of types of event triggers which can trigger a function. While different platforms differ to an extent in the type of events (many of them specific to other platform-specific services) which can used to trigger an event, a common list of popular options include:

- HTTP requests
- Insertion of item in queues
- Messages on topics in message brokers
- Timer based events
- Intent by digital assistants (Alexa Skills, Google Intents)
- Addition of item in a database
- Code build workflows such as master commits
- Applications logs belonging to a group

The increasingly ubiquitous nature of functions in cloud architectures also ensures a wider integration of functions across cloud services and expeditious support for integration with newer services as well.

### 1.1.5 Lightweight Virtualization for Serverless functions

In cloud infrastructure, functions and other types of workloads belonging to different customers maybe running on the same machine. Hence there is need of a strong isolation primitive to isolate workloads from each other. Isolation is also required to run the functions in separate environments without interfering with each other. The serverless functions are hence placed in such multi-tenant isolation primitives.

However, traditional isolation techniques such as virtualization cannot be used to place the serverless functions, primarily because they have a huge space overhead and have high start-up times. Instead, many lightweight virtualization techniques have been used to achieve the necessary isolation. These include containers, unikernels, library operating systems, etc. [8]

Open-source platforms such as Apache OpenWhisk [16] and OpenFaas [17] use docker containers to achieve their required isolation. Similarly, Google and Amazon have developed their own technologies such as gVisor[18] and Firecracker[19] respectively, to deploy the function instances on platforms.

## 1.2 Cold Start latency

One of the biggest challenges in adoption of serverless computing is the problem of cold start latency. In this section we will discuss the problem of cold start latency and the severity of the impact it can have in building web architectures.

### 1.2.1 Description

If an instance of a particular function is running, and a configured event is triggered, then the event is forwarded to the running instance and the function serves the event. If there are no instances of the function running, and a configured event is triggered, a new instance of the function is deployed and started. Another case a new instance for a function may be started is where there the currently running instances of the functions are being utilized up to their full capacity and the serverless platform decides to deploy additional instances to handle the incoming traffic.

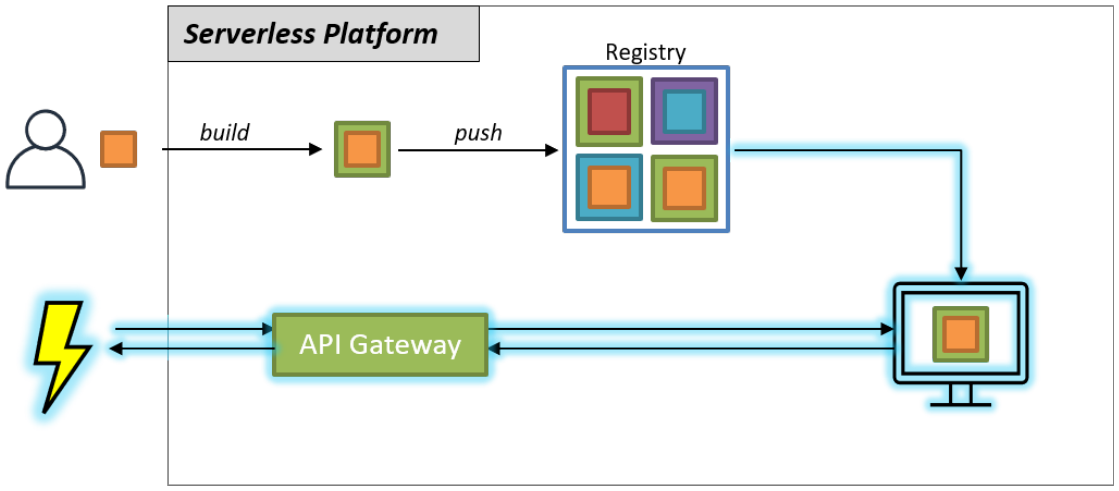


Figure 1.2: Diagram describing design of a serverless computing platform. Highlighted sections represent activations during cold execution.

When a new instance of a function is required to be deployed to serve an event, the response time of the event will be significantly higher compared to the case when the event is being served by an instance of the function which is already running. This delay will occur mainly due to the startup latency required to create the new instance of a function and is called the Cold Start Latency. The three important components of this startup latency include the time to deploy and start an instance of a cloud function, the time to initialize the environment for the code of the function to run and the time to initialize the function.[8]

In this thesis, executions which suffer from cold start latency are hereinafter referred to as **cold executions** and the remaining executions are referred to as **warm executions**. Figure 1.2 depicts the design of a serverless platform, where the highlighted sections refer to components activated during the cold execution.

### 1.2.2 Impact

It is important to understand the consequences of cold start latency to understand the reason it is poses such a big problem for their wider adoption of functions in web architectures. For one, cold start latency has a significant impact on the ability for developers to attain predictable performance [20], [8].Based on the application, the cold start latency even in a single event in a workflow may not be acceptable. As discussed in 1.2.3, the contributing factors to cold start latency may include the time to initialize a function, and functions with large start-up time may lead to time-out of events. This may have devastating consequences and may severely affect user experience[21].

We should also consider the fact that a single end to end workflow in a typical microservice based web architecture may consist of multiple functions in a chain. Hence, if a workflow is triggered after a while, the multiple functions in the workflow may all face cold start latency and hence increase the end to end response time significantly. Even if we consider that one function faces cold start at a time, the mere presence of multiple functions in the workflow may lead to multiple events getting slowed down due to multiple cold starts of different functions. We can also deduce that functions belonging to workflows sparsely invoked would suffer from a fate of encountering more cold starts. [22]

Another factor to remember is that the incoming traffic for a function may not have a equal distribution over time. Some applications may suddenly generate 'bursts' of traffic. Serverless platforms may invoke multiple function instances to handle the incoming traffic. Hence, a lot more events can be facing delay due to cold start latency at any given time. Some platforms such as AWS allow the option to set a concurrency limit to ensure that the amount of instances being created is not very huge. If the concurrency limit is reached, the remaining requests beyond the concurrency limit may be throttled and rejected, further alleviating the problem[23].

In order to avoid sudden scaling of a large amount of functions, developers may use techniques such as asynchronous invocation[24] or API gateways[25] to manage the traffic. While these techniques will definitely help avoid throttling, the events staying in queues still have to wait for the new functions to pick them up and hence face the same increased latency in response time.

### 1.2.3 Contributing factors

The cold start latency for different functions may vary based of a variety of factors. Some of these contributing factors [26], [27] include:

- Size of the deployment package
- Memory limit configuration set
- Programming Language : Static languages such as C# and Java have higher initialization times compared to dynamic languages like Python, JavaScript etc
- Third party libraries used [21]
- Function design and initialization costs [21]

## 1.3 Data Duplication

Our research work relies on the idea of avoiding the transfer of duplicate data from a server and reusing blocks of data from previously deployed instances instead. In this section we analyze the duplication of data in serverless computing and commonly used duplication strategies.

### 1.3.1 Data deduplication in Computing

Data duplication is a common phenomenon observed in data sources across computing. It is widely prevalent in storage systems companies where studies computed by Microsoft[28] [29] and EMC [30] [31] claim redundancy values to be as high as 50 to 80 percent. [32]. Redundancy can also be observed within a relatively smaller scope, where compression techniques such as LZ77/LZ88 [33] are prevalent to reduce size of data while transmission or storage.

Data deduplication presents the idea to reduce/ eliminate this redundancy by having single copies of data blocks instead of multiple. These help in saving storage costs and help in reducing network transmission costs, especially in low bandwidth networks.

While compression acts as a universal technique to reduce this deduplication, the algorithms generally reduce duplication across a window of few kilobytes.[33]. For deduplication across a larger scale, the strategy incorporated mainly involves dividing data into blocks (files or chunks) and linking the occurrence of duplicate blocks with references to the same copies. These references are generally based on cryptographic hashes which can be used to uniquely identify the data blocks. These reference are also used for indexing of blocks in databases to be quickly retrieved and served while accessing the original data source.[32]

### 1.3.2 Duplication in Serverless Computing

Serverless platforms provide templates for the users to write their code in designated sections called function handlers. Since the code written by the user is what really differs between functions of a serverless platform belonging to the same programming language, a significant amount of data belonging to the binaries of the functions could be duplicate. This may include the data files related to the web framework, the run-time framework (including the native libraries) as well the operating system libraries inside the package. Even among functions belonging to two different programming languages, the operating system libraries may be the same.

Also, while carefully inspecting the function handlers one may find a large amount of duplicate code among the various function handlers itself. The function handler code generally includes the code belonging to the user's business logic, the third party libraries used by the handler, configuration files as well as data files embedded in the code. The distribution of data among these components may be highly dependant on the type of application, as well as the programming language which the function belongs to. In languages such as Node.js, which rely heavily on micro-packages, a significant amount of third party libraries used by different functions may belong to a common set of most popular library packages.

## 1.4 Research Problem

### 1.4.1 Research Idea

Our research is based on the idea that serverless computing function samples belonging to multiple customers may have different pieces of code, but they have multiple blocks of data common within them. Each machine used for deployment simultaneously runs multiple such functions, and deploys newer instances at regular intervals of time. Hence there is a large amount of movement of data during each deployment, which includes redundancy due to commonality shared between newer functions and functions whose code has already been deployed on the machines. We could identify these common blocks of data, prevent their transmission to reduce the amount of bytes deployed, and hence reduce the transmission time.

### 1.4.2 Research Objectives

Our research objectives can be summarized to the following questions:

- **What is the extent of the impact of the size of the package on the deployment time of functions?**

  We run an analysis of the performance of serverless functions on a commercial serverless platform such as AWS Lambda. We vary the size of the package and observe a consistent increase in value of the cold start latency. We also run the test in our test setup using OpenFaas, and observe a similar correlation between package size and cold start latency values.

- **What is the amount of data shared between deployment packages belonging to different functions?**

  We inspect different deduplication algorithms to find similarity between deployment packages. We also generate a corpus of most popular library packages in Node.js and compare the deployment package of a function based on Node.js to find duplication.

- **Can we build the data for a function instance by re-using data blocks from previous deployments?**

  We analyze the data movement during deployment in our test setup, and enhance the deployment workflow to build a functioning prototype which incorporates our research idea to reduce complete transfer of data during the deployment to only a partial transfer. Remaining data is fetched from a local stash available on the machine selected for deployment.

- **Can we reduce the cold start latency by re-using data blocks?**

We analyze the performance improvements introduced by our design changes and demonstrate a reduction in the cold start latency.

### 1.4.3 Research Summary

In this thesis we describe our research to answer the above questions. In chapter 2 we recount the existing solutions for preventing cold start latency and discuss some of the current research work in this area. In chapter 3, we attempt to understand the commercial serverless computing platforms and the degree of impact of the package size on the cold start latency. In chapter 4, we discuss the design of OpenFaas and Docker Swarm, which we use as a reference architecture for implementing our prototype. In chapter 5 we aim to identify the potential for deduplication we can achieve, and develop strategies to achieve a good re-use rate. In 6, we present the design of our prototype by making changes to the reference architecture to improve the latency. In chapter 7 we present the results depicting the improvements we could gain, and discuss some metrics for a more practical deployment of our solution. In chapter 8, we summarize our work and discuss limitations and future work.

# 2. PREVIOUS WORK

## 2.1 Deduplication

Several algorithms have been developed over the years to improve deduplication at a large scale in storage systems. File level deduplication involves identifying common files and deduplicating the files.

Data streams can also be divided into blocks of data called as chunks. Existing solutions for chunking, mainly lie in two categories. The length of the blocks can be fixed size,where the chunking is agnostic to the content. Such a chunking strategy is called Fixed-size chunking (FSC). FSC however, introduces the danger of letting minor changes in one block affect the contents of all subsequent chunks in the stream. This problem is called the boundary shift problem. [32]

Chunking can also be of variable length, where the chunks are divided based on some knowledge of the content. This category is called as Content Defined Chunking. The size of the chunking windows are not always equal, and are instead terminated at breakpoints. Popular algorithms in this category include the Rabin Algorithm.

The chunked blocks are encoded to generate keys for indexing. The encoding algorithms such as sha256, are designed to generate unique keys (with a very low collision rate). These keys are used to identify duplicate blocks, as well as to refer them while retrieval.

## 2.2 Improving latency

Existing solutions to avoid cold start latency include:

### 2.2.1 Provisioned Concurrency

One of the most common techniques suggested by cloud providers to deal with cold start latency in functions is to provisioned concurrency. Provisioned concurrency mainly involves having a fixed set of instances of functions to be always running, to avoid the cold starts in workflows by always having warm functions running to serve events. Azure provides such a functionality under a ’premium’ plan where they have perpetually running warm instances or a ’dedicated’ plan where the customers can use their underutilized VMs to run warm instances of their functions.[34]. This also helps in giving developers predictable performance metrics.

While the idea of provisioned concurrency helps in avoiding the incidence of delays due to the cold start problem, it surely does not solve it. Cold start during bursts of traffic as mentioned in 1.2.2 can still not be avoided. Customers can chose to not use the provisioned scaling plans due to higher costs. Also, having provisioned concurrency hurts the idea of the truly “on-demand” nature of functions which is a big motivation for customers to chose the platform.

### 2.2.2 Periodic Warming

The technique of periodic warming is similar to provisioned concurrency, where instead of directly configuring a pool of warmed containers, certain dummy events are generated to warm functions in a timely manner. This helps in ensuring that the critical events related to the application run as warm executions. [22]

### 2.2.3 Pre-arrangement of resources

Many of the existing approaches try to improve the latency by pre-arranging the deployment environment. Some solutions suggest pre-creating and deploying containers which can be directly used during deployment. [35]

Mohan et al. [36], suggest improving the initialization times by pre-creating resources such as network interfaces. This method can help reduce cold start by upto 80 percent.

Akkus et al. [37] suggest making use of the requirement of weaker isolation boundaries required for functions belong to the same user. They propose re-using containers belonging to the same user for different functions owned by the user.

# 3. ANALYSIS OF COMMERCIAL SERVERLESS PLATFORMS

While cloud providers own state of the art technology in the serverless computing domain most of their code and architecture are proprietary and closed source. Such restrictions necessitate making use of techniques that rely on reverse engineering, in order to deduce to a certain extent, the internal properties of the serverless computing platforms. While there are several open-source offerings of serverless platforms in the domain, it is important to gather insights into the workings of the commercial, close-source serverless platforms, to validate our hypothesis and develop more realistic design choices.

## 3.1 Impact of deployment package size on cold start latency

In this thesis, we provide a mechanism to reduce the component of the cold start latency, which involves network transmission costs occurring due to the size of the deployment package size. Our research is based on the hypothesis that the size of the package has a significant impact on the cold start latency.

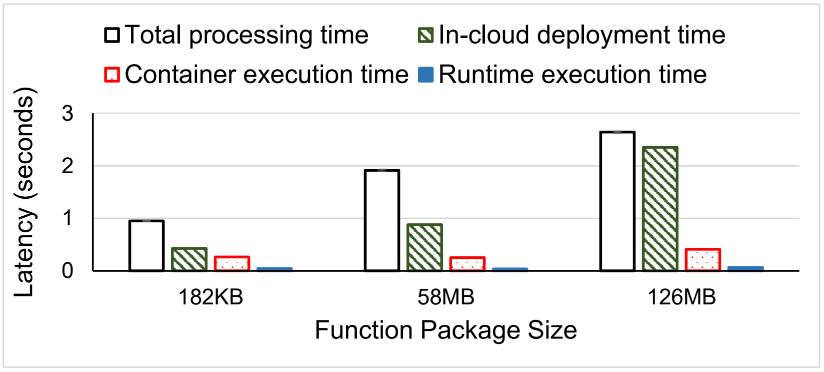


Figure 3.1: Variation of cold start latency with increase in package size

To verify our hypothesis, we conducted tests on the AWS Lambda platform. We run multiple lambdas of increasing sizes to verify the change in the response latency. The observed latencies in the executions have been presented in Figure 3.1. Our results have been reported for three package sizes: 182 KB, 58 MB, and 126 MB. AWS provides a telemetry service called the AWS X-Ray [38], which provides a breakdown of the execution time into components such as Initialization, Invocation and Overhead. Based on these components we could measure readings such as processing time, in-cloud deployment time, container execution time and the run-time execution time.

We can observe that the total processing time increases significantly as the size of the package was increased. On the other hand, latency values for warm executions remained almost in the same range of 15-30 milliseconds range, irrespective of the size of the package.

A notable challenge we had to overcome to run this test is the necessity distinguish between a warm execution and a cold execution. We make this distinction by referring to an identifier placed in a file in the temporary folder of the lambda. When the lambda serves its first event, the function handler has code that checks for the identifier file and reports its unavailability in the output of the function. Since the temporary folder is empty, the execution can be distinguished as a cold execution. The function handler further proceeds to generate a unique identifier and places it inside the identifier file. During warm executions, the file is found to be present, and the function handler reports its availability along with the identifier. The identifier serves as a mechanism to distinguish between multiple instances of the function.

## 3.2 Analysis of the function life cycle

We also carried out tests to analyze the life cycle of the function. The technique used was similar to that in 3.1, where we used the function handler code to analyze the environment running the function. We wrote a JavaScript function handler that executes shell commands. The results were reported in the response of the function handler.

One set of tests involved inspecting the *'/proc/time'* file handler, which reports the uptime of the environment running the function. The results have been provided in table 3.1. We observed that the uptime of the environment was significantly higher compared to the time the function was actually started. We carried out this test across the three major serverless platforms—AWS Lambda, Azure Functions, and Google Functions—and could observe similar results.

| Platform | Language | Uptime | Platform | Language | Uptime |
|---|---|---|---|---|---|
| AWS Lambda | JavaScript | 2678.94 s | Azure Functions | JavaScript | 969.36 s |
| AWS Lambda | JavaScript | 3254.36 s | Azure Functions | JavaScript | 133.84 s |
| AWS Lambda | Python | 2667.53 s | Azure Functions | Python | 218.00 s |
| AWS Lambda | Python | 782.48 s | Google Cloud Functions | JavaScript | 86.45 s |
| AWS Lambda | C++ | 1565.94 s | Google Cloud Functions | Python | 130.41 s |

Table 3.1: Table representing the time for which the function environments have been initialized before function deployment

Analyzing these results, we can deduce that environment running the function is started for a relatively long time before the function instance is deployed. Based on this observation, we can conclude that the serverless platforms *'pre-warm'* their run-time environments before deploying function packages to them.

| Platform | Language Runtime | Uptime |
|---|---|---|
| AWS Lambda | Node.js | 2676.78 s |
| AWS Lambda | Node.js | 3748.45 s |
| Azure Functions | Node.js | 967.69 s |
| Azure Functions | Node.js | 132.14 s |

Table 3.2: Table representing the time for which the language runtime has been running before function deployment

We performed another set of tests by inspecting the running processes inside the environment. We ran these tests to specifically trace the start time of the language runtime services, mainly Node.js for JavaScript handlers. The results of these tests have been provided in table 3.2. Our readings were similar to the previous experiment, where the start time of the runtime service was observed to be closer to the initialization of the runtime environment, and significantly in advance to the function deployment.

Hence, we can conclude that the virtualization environments, along with the language runtimes, are started well in-advance compared to the execution of the event. We can also conclude that environments are chosen to run handlers specific to a language beforehand, instead of it being a dynamic decision made during runtime.

# 4. OPENFAAS WITH DOCKER SWARM

## 4.1 OpenFaaS as a design choice

While serverless platforms are known to abide by a common set of core architectural principles, for the most part they are all developed separately by individual cloud vendors. Cloud vendors invest a large amount of money and resources into the developments of these platforms and have the state of the art in terms of technology.[39] Also, they have access to the high end infrastructure in their data-centers and real computing samples. Unfortunately, most of their code is closed source.

Hence, for the analysis and implementation of our research, we have to make use of Open Source platforms such as OpenFaaS and OpenWhisk[40, 17, 16, 41]. These platforms are popular choices as serverless platforms for companies that want to deploy their application workloads on their own infrastructure[42]. In our prototype, we make use of OpenFaaS. OpenFaaS is a popular serverless choice and it is a very active offering in this domain with 17.7K stars on GitHub.[43]

OpenFaaS provides the option between Docker Swarm[44] and Kubernetes[45] as its underlying container orchestration service. We chose Docker Swarm, primarily because its simple design and high code readability.

In the rest of this thesis, OpenFaaS is considered as the reference architecture for serverless platform. Our design choices are based on the design of OpenFaaS and our prototype is also based on making changes to the design of deployment as used in OpenFaaS. We intend to prove the functioning of our idea, and believe that other platforms can be appropriately modified to replicate our results.

## 4.2 Architecture

OpenFaaS provides a set of services which operate at different layers of the platform. It relies on docker containers to serve as the runtime environment for its functions and deploys them using container orchestration services such as Docker Swarm. Docker swarm takes care of the orchestration tasks such as deployment, load balancing, service management, etc.

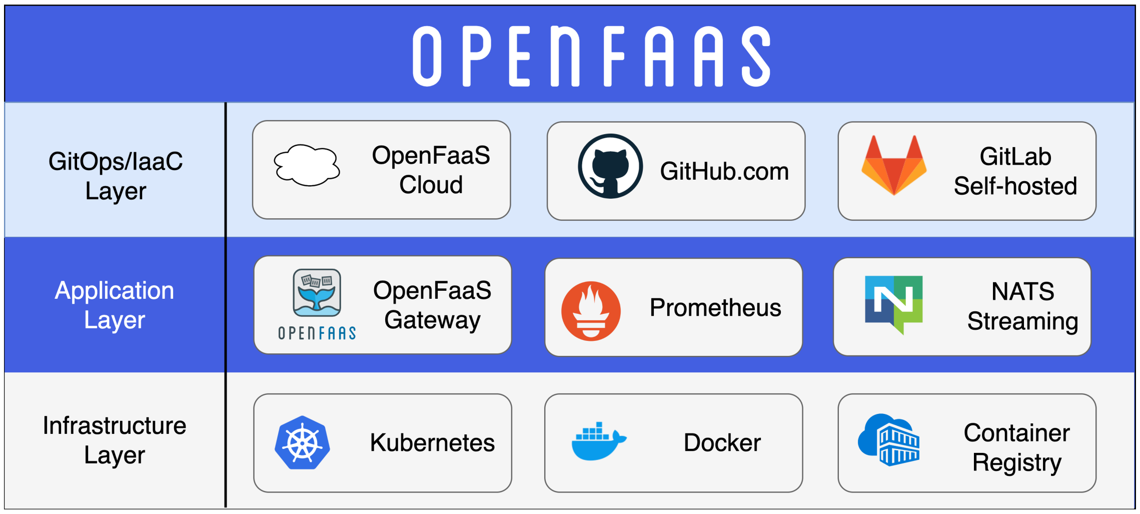


Figure 4.1: Overview of OpenFaaS reused from [1] under MIT license provided at [2]

Figure 4.1 provides an overview of the tooling in OpenFaas. OpenFaaS integrates its services with docker swarm by deploying them as docker containers itself. These docker containers are deployed using a 'docker-compose' file. They interact with the docker swarm cluster using the REST API exposed by the docker daemon.

### 4.2.1 OpenFaaS CLI

OpenFaaS CLI [46] is a command line utility which is used by the user to develop, integrate and deploy functions on the OpenFaaS platform. Some of the abilities of the OpenFaaS CLI we use in our prototype are:

1. **new**: The new command generates an empty function handler based on the template for language provided. The user can write the function code in the handler generated. It also generates a YAML file which can be used to configure the function properties such as event triggers.

2. **build**: The build command allows the user to generate a docker image out of the function written by the user(while using the ’new’ command).

3. **push**: The push command enables the user to push the function into a remote registry. This command in turn, pushes the docker image to the registry.

4. **deploy**: The deploy command is used by the user to deploy the function. The function will be deployed on the docker swarm cluster.

### 4.2.2 OpenFaaS Gateway

The OpenFaaS gateway[46] serves as a gateway service to handle the deployment of the OpenFaaS functions. It runs on all the manager nodes in the docker swarm and provides a REST API which is used by the CLI to interact with the user commands. The gateway also interacts with other services such as ’Prometheus’ and makes the decisions to scale the function instances.

### 4.2.3 Docker Registry

Docker registry [47] is a registry storage service which stores the docker images. It exposes a web interface using which, clients can run operations such as push and pull of images. The registry can either be a globally hosted service such as ’Docker Hub’[48] or can be a privately hosted service. A private instance of a docker registry can also be readily deployed as a docker container itself.

### 4.2.4 Docker Swarm

The docker engine can be configured to run in a swarm mode[44], where we set up multiple docker daemons on different machines in swarm mode to act as a distributed orchestration service. Each participant in a docker swarm can be configured run as a manager node or a worker mode. A manager node will also act as a worker node unless specifically prohibited from doing so. There can be more than one manager node, in which case a single manager node will get elected as a leader. The nodes in the docker swarm infrastructure are in communication using GRPC[49].

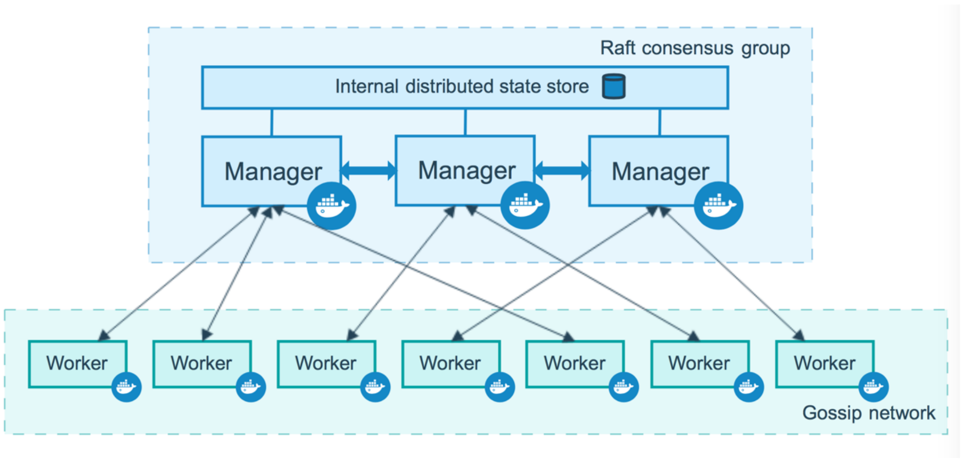


Figure 4.2: Docker Swarm Architecture reused from [3] under Apache license provided at [4]

The distributed consensus and leader election among the various nodes is handled by the RAFT protocol.[50]

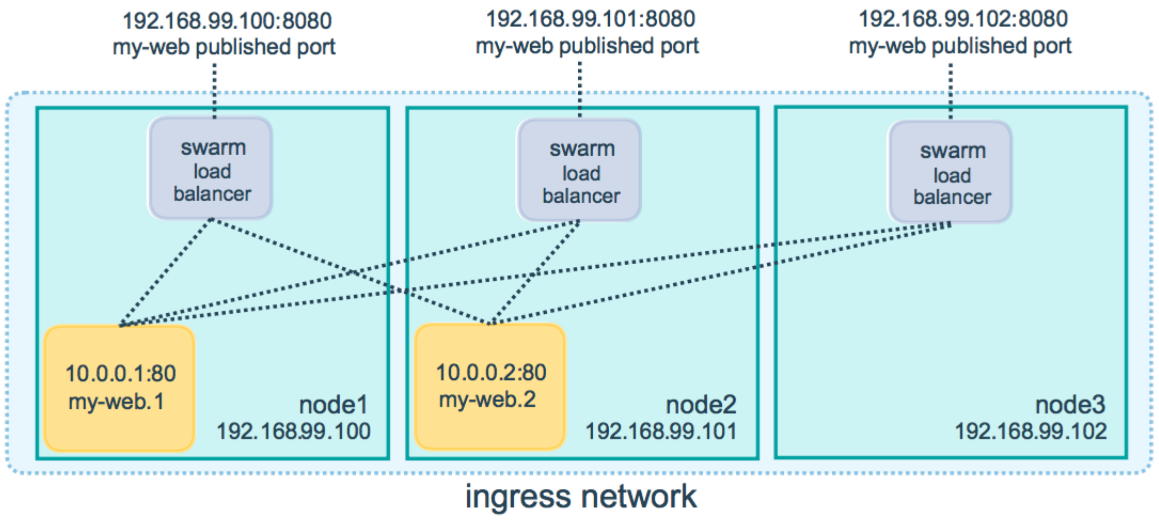


Figure 4.3: Router mesh network and load balancer in Docker Swarm architecture reused from [3] under Apache license provided at [4]

The functions are deployed on the infrastructure as services. An ingress swarm load balancer takes care of load balancing in the swarm architecture by generating a routing mesh. Hence, irrespective of whether a node holds an instance of a function, a request on any of the nodes in the swarm, would be redirect to a running function instance.

Figure 4.2 depicts the overview of a distributed system arranged to run docker swarm, and figure 4.3 depicts the routing and load balancing inside the distributed network.

### 4.2.5 Docker Image

A docker image serves as a read-only template, which, when run can be used to create a docker container. A container is hence a running instance of a image, similar to the 'class-object' relationship in object-oriented programming.

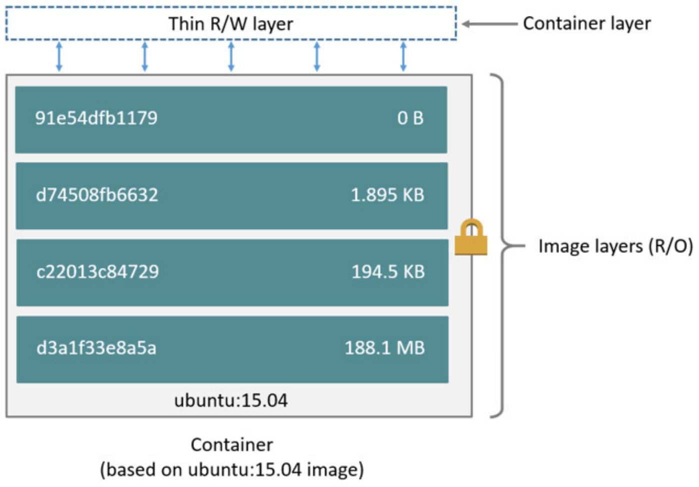


Figure 4.4: Docker image layers reused from [5] under Apache license provided at [4]

A docker image is generated from a script file called as the DockerFile[51]. When a docker image is being generated, the files involved in the execution of each line in the script gets converted to a layer on the file system. For e.g. a line in the DockerFile which installs the Java Development Kit (JDK), would get converted to a layer in the file system which contains all the files which get installed in the file system after JDK installation. These layers get stacked to get organized in a union mount filesystem[52]. The layers are read-only. When docker creates a container from an image, it adds a thin writable layer on top, in which all the files modified or created during the lifetime of the container is maintained.

A docker image also additionally contains a configuration file called Manifest which contains the metadata information as well as information to setup the environment.

## 4.3 Function Deployment in OpenFaaS

When a function is to be deployed on the docker swarm infrastructure, the gateway communicates this information to a manager node in the docker swarm architecture. While a lot of the management services in docker swarm can be carried out by any of the manager nodes, function deployment is specifically carried out by only the master node. The deployment intent is hence communicated with the docker swarm master, which selects a worker node for placement. The worker node is then requested to pull the docker image and start the container.

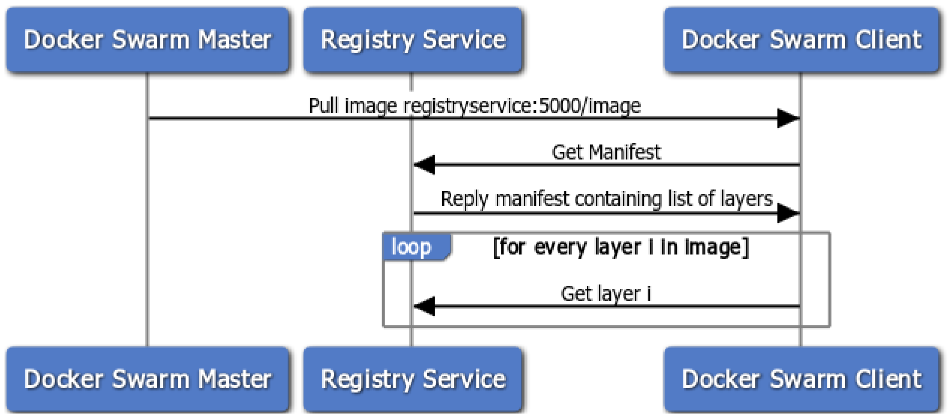


Figure 4.5: Worklow to pull a docker image

We can observe that the transmission of the function instance during deployment in OpenFaaS is carried out by the “image pull” workflow in docker. The image pull workflow is summarized in 4.5. The pull request provided to the client node consists of the domain address of the registry service hosting the image along with the image name/identifier. The image pull is carried out in two steps: first the manifest is fetched via a HTTP request, and then by using the ids of the layers found in the manifest, the individual layers are requested iteratively.

# 5. DEDUPLICATION IN SERVERLESS COMPUTING

The central idea of this thesis is to save transmission costs by only transferring data for blocks that are not locally available on the target node. In this chapter, we discuss the potential sources of data deduplication in our function instances, the strategies to chunk, and sources of data used to deduplicate.

## 5.1 Design for deduplication with OpenFaaS

As discussed in Chapter 4, we primarily focus on OpenFaaS on Docker Swarm as our reference architecture for providing our design suggestions. We know that the function instance is wrapped into an image and transmitted through the network. The data inside the function is organized into layers. During the storage of data inside the registry and transmission over to the client node, the files belonging to the layers are archived into tar files and compressed using gzip. Each layer is hence treated as a byte stream.

### 5.1.1 Data sources for deduplication

When an image is deployed to a client machine, the image layers are broken down into blocks of fixed sizes. These blocks are stored in a database attached to the client, which we call the 'stash.' Additionally, we deploy a global service that maintains a corpus of most used blocks globally. We also run a synchronization service that synchronizes this global corpus of blocks with the client stash periodically. Figure 5.1 depicts the various sources of deduplication for serverless functions on the client machine.

When a newer image is attempted to be deployed on the same client machine, the blocks which are not available in the client stash are not moved over. A more elaborate design explaining the complete workflow for achieving the above has been provided in chapter 6.

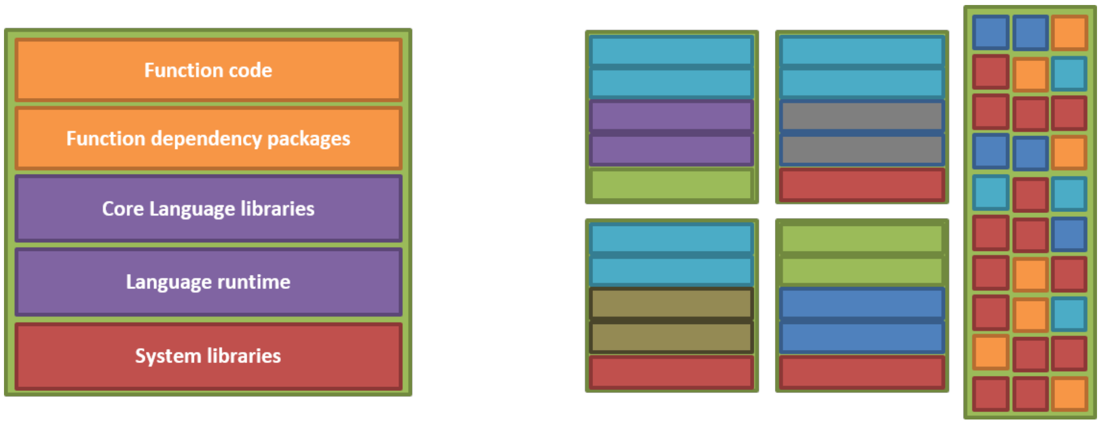


Figure 5.1: Diagram representing a sample function to deploy, and sources of deduplication in the client stash including previously deployed functions and global corpus of most used data chunks

### 5.1.2 Deduplication of layers

Docker has an in-built layer deduplication technique, where, on receiving the manifest, the docker engine on the client starts checking the layer ids of the image. Beginning from the bottom most layer, the client checks, if the layer in the image to be pulled, is already available on the machine. It runs this check using the sha256 id of the layer found in the manifest. If the layer is available, it skips the transmission of the layer. It continues this trend until it finds the first layer, which it does not find to be available in the client machine. All the layers above this layer are selected to be transferred over the network.

While this layering approach helps is deduplicating the common lower layers between the docker image being pulled, and the images already available, using our method, we provide an additional opportunity to re-use data from previous layers. In the deduplication strategy provided by docker, once a layer on the bottom is found to be not available on the client, even if a layer above this unique layer matches with the layers available on the client-side, the data is still transmitted. In our approach, however, if the client finds blocks belonging to a layer in its stash, it deduplicates it.

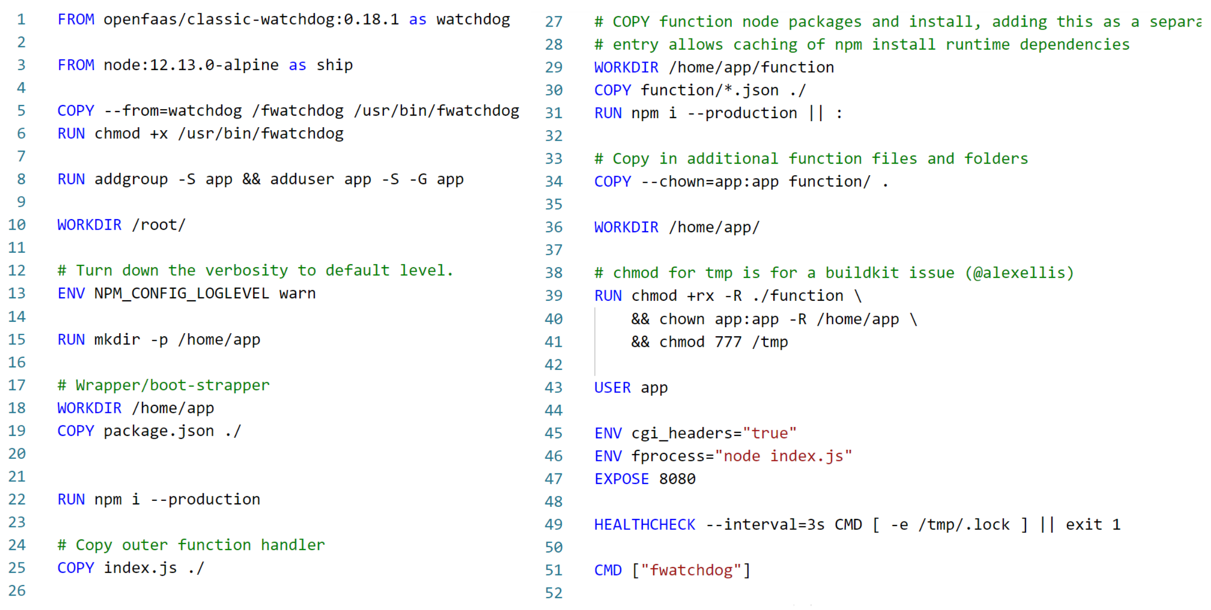
### 5.1.3 Deduplication of data blocks within layers

Additionally, while two layers are not exactly the same, there is plenty of opportunity for a good share of data blocks to be the same. For instance, if the layer contains library packages of the function, then although the function may have a unique set of library dependencies, there is a high chance of incidence of a majority of the blocks belonging to the library files in our client stash. A function may be using a good amount of popular libraries, which are highly likely to be available in the client stash from either the global corpus or previous deployments.

Another case where there can be high amount of re-use of blocks from the client stash, is when another version of a software has been deployed as part of a previous function on the same machine. In this case, while the layer matching would fail, deduplicating at a file-level would help ensure that a significant amount of files that are the same among the different versions are deduplicated.

### 5.1.4 Analysis of deduplication in a sample openfaas function

In this section, we will look at the docker image belonging to a function in OpenFaaS to understand the various layers and identify the potential for deduplication.



```
FROM openfaas/classic-watchdog:0.18.1 as watchdog

FROM node:12.13.0-alpine as ship

COPY --from=watchdog /fwatchdog /usr/bin/fwatchdog
RUN chmod +x /usr/bin/fwatchdog

RUN addgroup -S app && adduser app -S -G app

WORKDIR /root/

# Turn down the verbosity to default level.
ENV NPM_CONFIG_LOGLEVEL warn

RUN mkdir -p /home/app

# Wrapper/boot-strapper
WORKDIR /home/app
COPY package.json ./


RUN npm i --production

# Copy outer function handler
COPY index.js ./

# COPY function node packages and install, adding this as a separa
# entry allows caching of npm install runtime dependencies
WORKDIR /home/app/function
COPY function/*.json ./
RUN npm i --production || :

# Copy in additional function files and folders
COPY --chown=app:app function/ .

WORKDIR /home/app/

# chmod for tmp is for a buildkit issue (@alexellis)
RUN chmod +rx -R ./function \
    && chown app:app -R /home/app \
    && chmod 777 /tmp

USER app

ENV cgi_headers="true"
ENV fprocess="node index.js"
EXPOSE 8080

HEALTHCHECK --interval=3s CMD [ -e /tmp/.lock ] || exit 1

CMD ["fwatchdog"]

```

Figure 5.2: Docker file for a Node.js function template

A breakdown of the dockerfile for a sample function provided in figure 5.2, can be observed to be as follows:

- Lines 1 - 3 refer to bases images from which the image is derived. These layers are generally heavy. Line 3 would contain the Node.js runtime as well as the base libraries. While all images in openfaas belonging to Node JS follow the same template, templates belonging to other languages still refer to the base image referred to in Line 1. Hence, there is a high chance that the data in these layers are already present on the client machine, as part of a previous deployment. Hence these layers can be easily deduplicated.

- Lines 5-6 work on copying certain binaries to the docker instance. Lines 8-18 consist mainly of operations carried out to set up the environment. The data corresponding to these layers is mostly empty. Line 19 copies the package.json for the base image, and Line 22 installs these packages. Line 25 copies the outer js file as part of the openfaas to the inside of the docker. The files in these layers can be deduplicated if a Node.js function has been previously deployed on the same machine.

- Line 29-30 involves copying the 'package.json', which contains a list of third party library dependencies belonging to the function handler written by the user and installs these dependencies. We cannot expect all functions to use the same set of libraries; hence these layers as a whole are assumed to be unique for each function. However, a lot of these library packages are bound to belong to a global set of most used packages and would have probably been used before in other functions.

- Lines 34-39 involve copying the function handler code to the inside of the image. While they may share blocks of data with other functions, we cannot make any assumptions here. Apart from the function handlers, there may be additional files such as data files or configuration files. For example, if the function serves a web page, it may carry some images such as icons. If these images are popular, we may find some re-use of these data blocks.

- Lines 39-52 involve the setup of the environment and do not include much data. The layers corresponding to these lines are mostly empty.

## 5.2 Chunking of data

As discussed in 5.1.3, we can achieve a reasonable re-use rate only when we identify the re-use of data blocks within the layers by matching them with blocks of data in the client stash. For finding a reasonable re-use rate, we need to chunk the layers appropriately. The size of the chunks should also be optimal to ensure that they have the granularity to identify higher re-use while at the same time, it should be large enough to ensure that the number of chunks does not lead to excessive overhead.

### 5.2.1 Sliding window approach

We explored a sliding window strategy where we held a window of fixed size, around 4KB, and slid the window each time with an offset. After each shift, the contents of the window were hashed to form a key for matching. [32]

For comparing two functions, while chunking the first function, we maintained a set data structure in which we added the hashes of our chunks. While chunking the second function, we checked the availability of the chunks in the set data structure to measure the re-use rates.

#### *5.2.1.1 Results*

We ran multiple iterations of the experiment with multiple block sizes and offset sizes but could not find a significant re-use rate. The results of our tests have been provided in table 5.1. For measuring the effectiveness of our strategy to identify duplication, we built two functions in Node.js, say sample functions 1 and 2, of sizes 118 MB and 143 MB respectively.

#### *5.2.1.2 Analysis*

On analysis of our approach, we could conclude that even though a significant amount of files among these layers would be common, the data would be organized differently. We observed that the beginning of the chunking window would not be aligned with the beginning of the byte stream

| Block Size | Offset | Re-use rate |
|---|---|---|
| 500 | 250 | 0.75 % |
| 1000 | 500 | 0.41 % |
| 2000 | 1000 | 0.277 % |
| 4000 | 2000 | 0.07 % |

Table 5.1: Table representing re-use rates with sliding window approach

of the files inside the layers. Hence due to the varying offsets, the same file would be chunked differently in both layers and blocks the would not match.

### 5.2.2 Content aware chunking

While our initial approach of finding an ideal chunking size was agnostic to the content of the byte streams of our layers, as explained in section 5.2.1, we could observe that due to the variation in file organization, an agnostic approach to the organization of files inside the layers would lead to us to not finding an appropriate re-use rate.

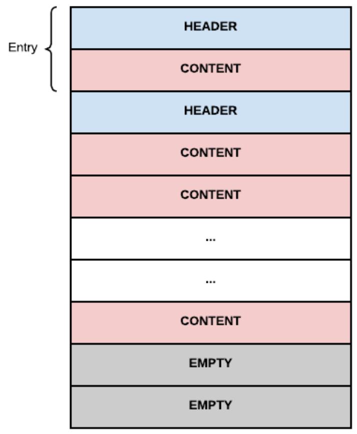


Figure 5.3: Organization of files inside a tar reused from [6] under license provided at [7]

#### 5.2.2.1 *Size of chunk*

To make sure that our chunks are aligned with the start of the files inside the tar we exploited the design of file organization inside a tar. Figure 5.3 demonstrates the organization of files inside a tar. A file is preceded by metadata of around 512 bytes. The files are also rounded off to a multiple of 512 bytes by adding padding of zeroes at the end. Also, the tar is ended by two empty blocks of 512 bytes. Hence we can observe that contents of tar are organized in multiples of 512 bytes. [53] Consequently, upon dividing files into fixed chunks of 512 bytes, we can observe that the beginning of the chunks are aligned to the start of the files. Thus we could find a significant rate of re-use.

#### 5.2.2.2 *Results*

For measuring the effectiveness of our strategy to identify duplication, we built two functions in Node.js, say sample functions 1 and 2, of sizes 118 MB and 143 MB respectively. We only compared the layers corresponding to the function handler and its dependant libraries since the rest of the layers are the same and hence have a 100 percent re-use rate. We chunked the sample function two and added the hash of the chunks into a set. Sample function 1 was also chunked and the hash values of these chunks were checked against the set to determine if they were being re-used or not.

| Deduplication source | Size | Block Size | Re-use rate |
|---|---|---|---|
| Sample Function 2 | 143.74 MB | 256 Bytes | 31.83 % |
| Sample Function 2 | 143.74 MB | 512 Bytes | 28.46 % |
| Sample Function 2 | 143.74 MB | 1024 Bytes | 13.11 % |
| Top 200 node modules | 507.54 MB | 512 Bytes | 63.43 % |
| Top 1000 node module | 1767.37 MB | 512 Bytes | 68.55 % |

Table 5.2: Rate of deduplication observed by comparing the function handler layers of sample function with other sources

Additionally, we created a corpus of the top 200 and the top 1000 most used packages in

Node.js[54], ranked by popularity. The chunks from the sample function were hence checked against hash sets generated from these corpora to determine a re-use rate.

We can also observe that we can alternatively chose to use block sizes of 128 bytes or 256 bytes instead of 512 bytes. Choosing these values would still help us ensure that the start of the chunks get aligned with the start of the files. From 5.2 we can also observe that using 256 byes would increase the re-use rate as it would help us identify partial matches among files. However, there is a trade-off associated between using smaller blocks for higher re-use rate and the overhead due to increase computation and I/O costs associated with a large amount of blocks. Considering both options, we would chose 512 bytes as an ideal size of the chunking window.

# 6. DESIGN

In this chapter, we discuss the changes we made to the design of the docker pull workflow to develop a prototype which demonstrates a reduction in cold-start latency when there is the re-use of blocks.

## 6.1 Chunking of layers

As discussed in chapter 5, we chunk our layers to 512-byte blocks. The chunking is done during the docker push workflow, when the images are pushed to the registry for hosting. Upon chunking and creating the fingerprints from the chunks, a digest of the generated fingerprints is created. This digest is termed as the *recipe*. A recipe is created for each layer in the image and inserted into a DB instance provisioned in the environment hosting the registry service.

## 6.2 Function deployment

In section 4.3, we discuss the workflow to pull a docker image. The client first fetches a manifest from which it retrieves a list of layers to download. These layers are identified by their ids. The client then requests all the layers iteratively using HTTP requests.

We provide two alternate designs to modify the retrieval of the layer as part of the image pull workflow. The designs differ primarily in terms of the knowledge the registry server has of the contents of the client stash. Therefore, since in the stateless design, the registry server does not hold client-state information, it has been termed as stateless. Similarly, in the stateful design, the registry service relies on knowledge of the contents of the client stash.

### 6.2.1 Stateless design

In the stateless design, the registry service has no information on the state of the client stash. Hence, before beginning the pulling of layers, the client fetches a collection of the layer recipes from the registry service.

To pull a layer from the registry service, the client first has a look at the fingerprints of the layer chunks in the registry service. It verifies whether the chunk is available in the client stash. Based on whether the fingerprint is available or not, it creates a bitstream of 1s and 0s, where each bit represents a chunk in the recipe. We call this bitstream “declaration”.

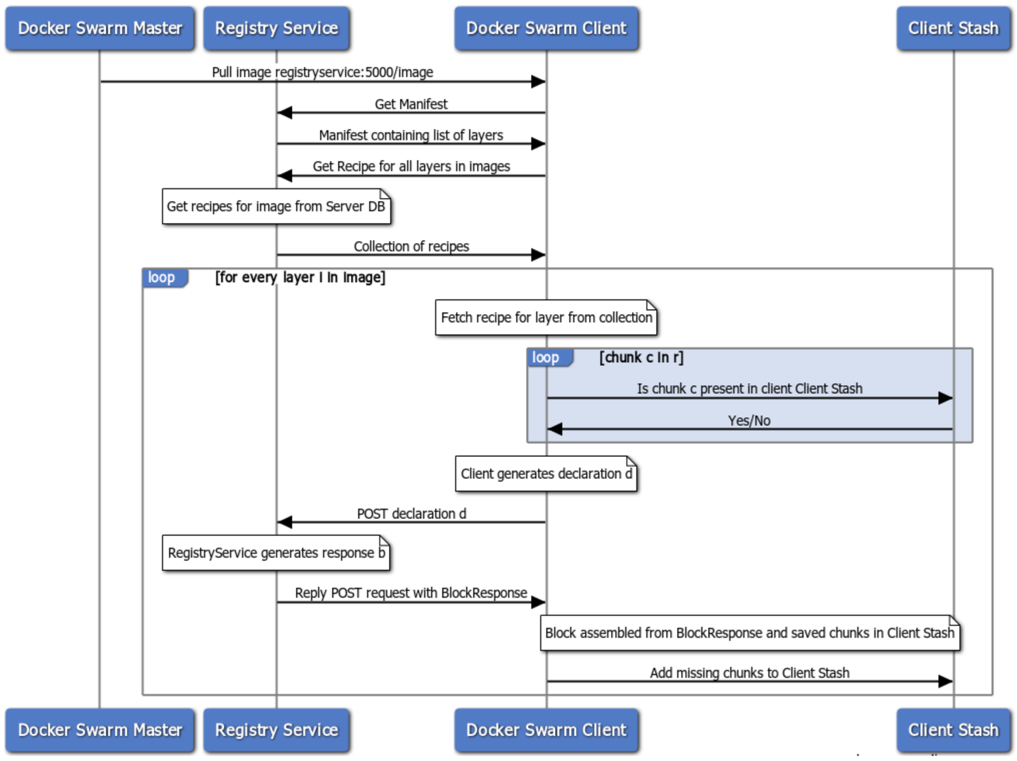

Docker Swarm Master
Registry Service
Docker Swarm Client
Client Stash
Pull image registryservice:5000/image
Get Manifest
Manifest containing list of layers
Get Recipe for all layers in images
Get recipes for image from Server DB
Collection of recipes
loop
[for every layer l in image]
Fetch recipe for layer from collection
loop
[chunk c in r]
Is chunk c present in client Client Stash
Yes/No
Client generates declaration d
POST declaration d
RegistryService generates response b
Reply POST request with BlockResponse
Block assembled from BlockResponse and saved chunks in Client Stash
Add missing chunks to Client Stash
Docker Swarm Master
Registry Service
Docker Swarm Client
Client Stash

Figure 6.1: Workflow for stateless design

Next, replacing the call to fetch the blob in the original docker workflow directly, the client posts the declaration to the registry service. The registry service reads this declaration and generates its response, where it only sends the chunks corresponding to the 0s in the declaration, which indicates that the client does not have the chunk.

Once the client receives the response from the server, it assembles the layer using the chunks received from the server and the remaining chunks available in the database.

Also, once the retrieval of the layer is done, the client service adds the chunks obtained from the registry service into the client stash. This provides the client with the ability to deduplicate chunks as part of future deployments using the data transmitted during the current deployment.Figure 6.1 depicts the workflow in the stateless design.

### 6.2.2 Stateful design

In the stateful approach, a synchronization service pre-synchronizes the state of the client stash with the server. With this synchronization, the server is aware of the ids of the chunks present on the client. Since only the ids of the chunks in the client stash are synchronized with the server DB, the storage required for storing this information is relatively low.

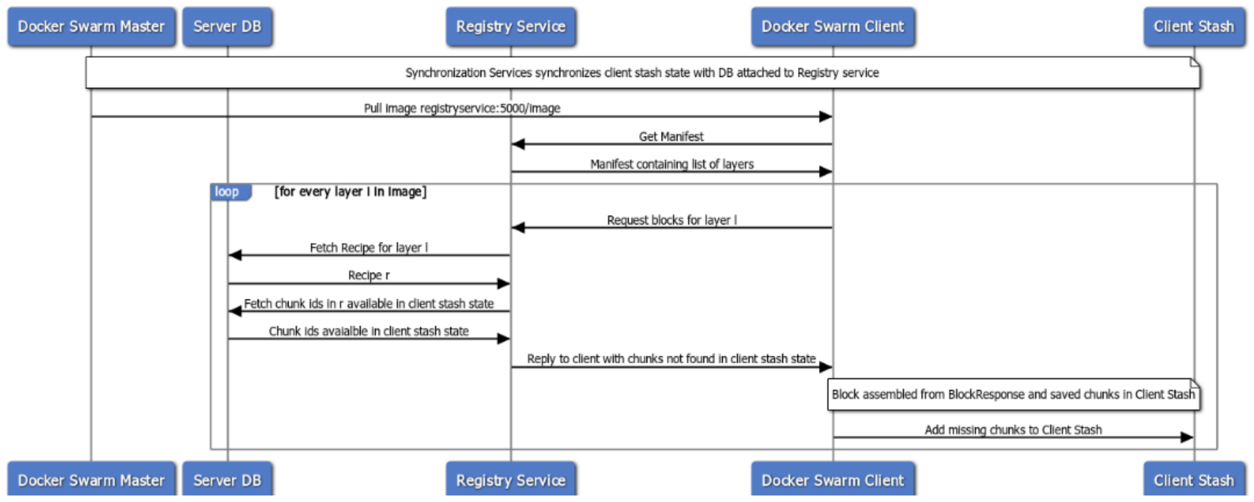

Docker Swarm Master
Server DB
Registry Service
Docker Swarm Client
Client Stash
Synchronization Services synchronizes client stash state with DB attached to Registry service
Pull image registryservice:5000/image
Get Manifest
Manifest containing list of layers
loop
[for every layer l in image]
Request blocks for layer l
Fetch Recipe for layer l
Recipe r
Fetch chunk ids in r available in client stash state
Chunk ids avaialble in client stash state
Reply to client with chunks not found in client stash state
Block assembled from BlockResponse and saved chunks in Client Stash
Add missing chunks to Client Stash

Figure 6.2: Workflow for stateful design

When the client requests for a layer, the registry service fetches the recipe for the layer from the server DB. It parses the chunk ids in the recipe document and checks with the state information of the client stash. Based on the availability of chunks, it determines which chunks it needs to send to the client. For the rest of the chunks, it just sends the ids.

On the client-side, the response from the registry service is parsed. For the chunks, whose identifiers have been transmitted instead of the actual chunk content, are retrieved from the client stash. The layer is consequently reassembled based on the chunks gathered from both sources. Figure 6.2 depicts the workflow in the stateful design.

## 6.3 Chunk Ids

We need an effective fingerprinting mechanism to generate unique keys as identifiers for the chunks. These fingerprints must be unique since they are used for identification of deduplication and indexing inside a database.

### 6.3.1 Fingerprints of chunks

We instead use the MD5 [55] algorithm to generate a hash of the block, which is used as the fingerprint. The MD5 algorithm generates a hash of around 128 bits. Although, SHA256[56] is a popular choice as a solution for such requirements, which generates a key of 256 bits, since our performance improvements rely heavily of reducing the amount of bytes transmitted, we need smaller fingerprints.

While using as an identifier in our database, or for formatting during transmission, we need a representation of the fingerprint key value in terms of a string. To achieve this, we use the base64 algorithm for encoding. This encoding scheme will generate 22 characters for the 128 bits of hash value.

### 6.3.2 Global encoding scheme

As discussed in chapter 5, we have two sources of deduplication in our client stash. The chunks from deployments are stored in the client stash for deduplicating blocks for future deployments. Additionally, we have a global corpus of most used chunks, which we synchronize periodically with our client stash. This global corpus is created dynamically by accumulating data on frequency of usage of blocks from multiple clients on a global scale. The frequency values are maintained globally in a frequency table service like Amazon s3[10].

A batch process will periodically look into the frequency table, sorted by frequency, and select a limit based on the size limitations of the global database. All the chunks above this limit are selected to be included in the global corpus. A Huffman tree is used to generate the keys of the chunks in the global corpus, thus building a global encoding scheme. Since the frequency table would keep changing, each version of the global encoding scheme is marked with a version number.

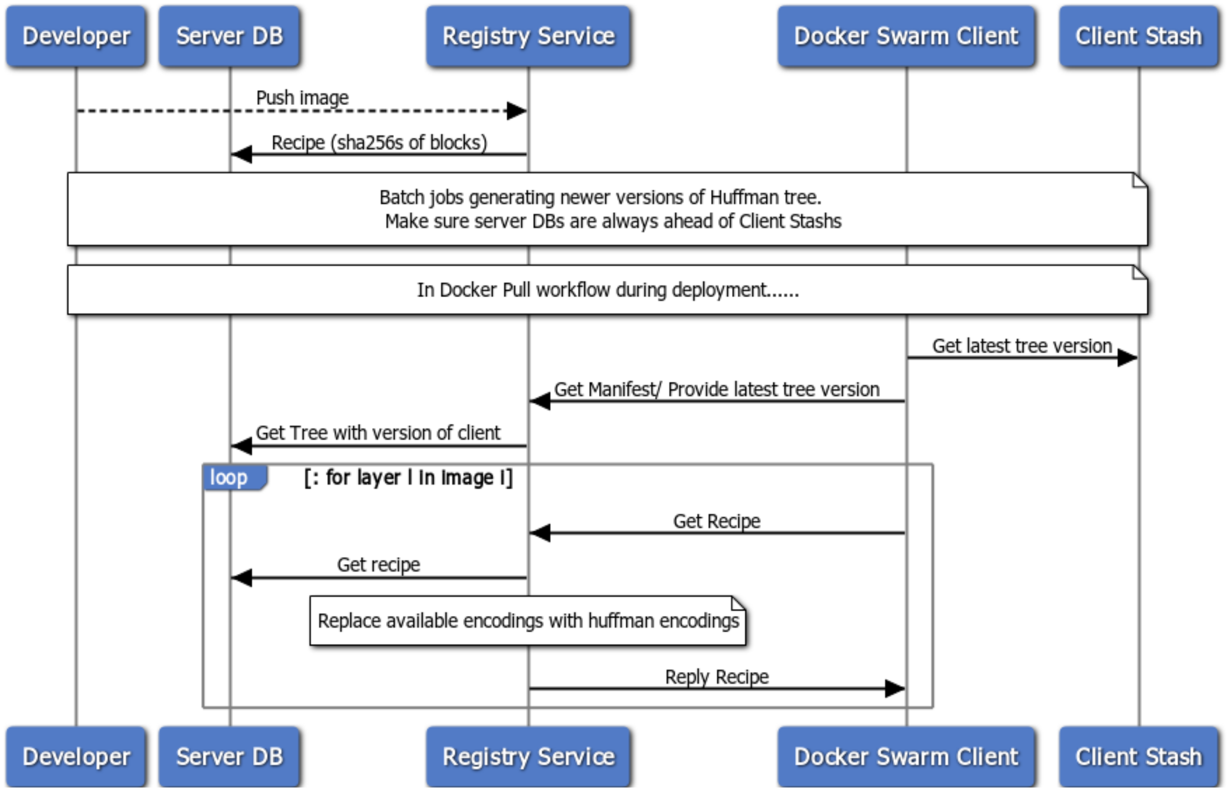

Developer
Server DB
Registry Service
Docker Swarm Client
Client Stash
Push image
Recipe (sha256s of blocks)
Batch jobs generating newer versions of Huffman tree.
Make sure server DBs are always ahead of Client Stashs
In Docker Pull workflow during deployment......
Get latest tree version
Get Manifest/ Provide latest tree version
Get Tree with version of client
loop
[: for layer l in image I]
Get Recipe
Get recipe
Replace available encodings with huffman encodings
Reply Recipe

Figure 6.3: Integration of global encoding scheme with stateless design

A synchronization service will periodically sync the global corpus with the client stash. The database connected to the registry service is updated with the global encoding scheme as well. Since the global scheme is dynamic and changes periodically, there can be an inconsistency between the encoding scheme of the blocks in the client and the server. This is acceptable since our design allows for a weak consistency model for synchronization with the global scheme.

Instead, we have to make sure that the registry service servers are synchronized with the newer versions of the encoding scheme before the client versions are. Since the server only needs to maintain the keys of the scheme instead of the actual blocks, it can hold more information on the past few versions of the encoding scheme with significantly lesser space requirements. While making the initial transactions during the image pull workflow, the client can notify the server of the version of the global encoding scheme it holds. The server can refer to a previous version of the global encoding scheme based on the client scheme version. When the server and the client agree on a specific version number, the server can replace all the chunk ids available found in the encoding scheme with their corresponding values in the scheme.Figure 6.3 depicts the proposed design changes to incorpoate a global encoding scheme.

## 6.4 Verification of layer

As discussed in section 6.3, we use the MD5**??** checksum algorithm to generate the fingerprint of the chunk. This fingerprint is used as an identifier to index the database. Since md5 generates a hash of only 128 bits, it is weaker than its counterparts, such as sha256 in terms of collision resistance.

Hence there is a very little possibility, more theoretical than practical, that with the use of md5, two chunks are identified to be the same because they have the same md5 fingerprint. This may lead to a wrong layer being reconstructed. However, we still need to safeguard our design against such an error.

Also, there may be some error during transmission of the bits, which we may need to verify.

To fulfill this requirement, once the layer is generated, we generate a sha256 of the entire layer byte stream. This is matched with the sha256 of the original blob transmitted from the server. In case these values do not match, we can identify the presence of an error due to the collision of checksum values of the two blocks, or a possible transmission error. In either case, we repeat the fetching of the layer while substituting the modified pull workflow with using the original docker pull workflow. The chunks from this transmission are not inserted into the client's stash.

## 6.5 State Synchronization Service

For the stateful design, we need a synchronization service which could effectively synchronize the state representation of the client stash present in the server. Requiring a strict consistency among the client stash and this state representation may add significant consistency and scalability problems, thus making our design inconceivable. Instead we propose a weak consistency model between the client stash state representation with the registry service and the actual state of the client stash, exclusively pertaining to addition of blocks in the client stash. Hence, if the representation of the state of client stash is missing some blocks which are present in the client stash, it is acceptable. This is because, if some blocks are not available in the client stash state, the registry service will just send them over.

However, if the client stash does not have some blocks which are assumed to be available in its representation of the client stash, this may lead to failure in successful transmission of the service. Such a case can occur only when chunks are being removed from the client database during some cleanup. We can ensure such that every time such a cleanup occurs, we increment the version of the client stash. Hence if there is a mismatch in the version of the state representation of the client stash and the actual version of the client stash, we do not use it and either chose a different machine for function placement, or resort to the original docker workflow for data movement.

Also, the need of a client stash state with the registry service adds some limitations on the machines the registry service can deploy to. The registry service is compelled to deploy only to machines whose state is available with it. We hence need to have multiple copies of the function distributed across multiple registry services, in-order to ensure that a wide range of machines options are available to chose the placement of the services.

# 7. EVALUATION

In this chapter, we discuss the evaluation of our proposed solution. As described in chapter 6, we propose two designs as, one being stateless and the other being stateful. We implemented these designs on our test setup and present the solutions in section 7.5.

## 7.1 Test setup

Our test setup consists of a docker swarm cluster set up over two machines- Machine A and Machine B. The configuration of these machines is the same and has been provided in 7.1. Machine A is assigned the role of manager and Machine B, the role of a worker/client node. We configure OpenFaaS over this docker swarm cluster. Machine A also hosts the registry server and contains the images of the functions. We use a redis[57] database for our server, as well as the client stash. Our machines are very closely located and may not represent network latency in a more realistic scenario. Hence, we slow down the network, by introducing latency of around 50 milliseconds.

| Parameter | Value |
|---|---|
| Model | Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz |
| Num of cores | 6 |
| RAM | 16 GB |

Table 7.1: Specification of machines used in test setup

## 7.2 Test samples

For our tests, we consider functions of varying sizes. These sample functions have been written in Node JS. The details of these sample functions are provided in table 7.2.

## 7.3 Test methodology

For running our tests, we deploy the sample functions using the OpenFaaS CLI. The performance is measured using logs in the code. For an image, the pulling of the layers happens concur-

| Function S | Function M | Function L | Function XL |
|---|---|---|---|
| 96 MB | 220 MB | 345 MB | 470 MB |

Table 7.2: Table representing the labels of the sample functions used along with their sizes

rently via various threads. However, once the layers have been pulled, the docker daemon takes time to set up the layers on the file system. It uses a synchronization primitive, to make sure the layers get set up individually and does this task serially. We hence report two readings, the time to download the layers and the time to pull the image.

We report our results for different percentage values of deduplication. Since we cannot achieve fine re-use values using real samples, we employ a technique where we deploy a function, and purge the client stash to a specified value, manually before re-deploying it. This helps us achieve precise re-use rates for our experiments.

## 7.4 Performance of Image pull in Docker Swarm

We first measure the performance of pulling an image in docker flow. The results are presented in table 7.3.

| Function Label | Layer download time | Total image pull time |
|---|---:|---:|
| Function S | 1.538 s | 9.607 s |
| Function M | 2.379 s | 10.921 s |
| Function L | 2.722 s | 11.562 s |
| Function XL | 2.912 s | 13.561 s |

Table 7.3: Table representing time to download all layers and time to pull image

## 7.5 Performance and Analysis of proposed solution

In table 7.4 and 7.5, we present the performance of the time to pull an image. For each variation of function size, we present a set of 3 readings, the performance of the original docker pull workflow, compared with the performance of the modified workflow when there is a 0 percent re-use rate and a 100 percent re-use rate.

| Function S | Layer download time | Percentage change | Total image pull time |
|---|---|---|---|
| Docker workflow | 1.538s s | - | 9.607 s |
| 0 percent re-use | 2.347 s | +52.28% | 10.094 s |
| 100 percent re-use | 1.827 s | +18.79% | 9.264 s |

| Function M | Layer download time | Percentage change | Total image pull time |
|---|---|---|---|
| Docker workflow | 02.379 s | - | 10.921 s |
| 0 percent re-use | 03.277 s | +37.73% | 10.548 s |
| 100 percent re-use | 2.057 s | -13.54% | 10.225 s |

| Function XL | Layer download time | Percentage change | Total image pull time |
|---|---|---|---|
| Docker workflow | 02.2912 s | - | 13.561 s |
| 0 percent re-use | 4.075 s | +39.92% | 13.342 s |
| 100 percent re-use | 2.775 s | -4.72% | 13.456 s |

Table 7.4: Table representing performance of function deployments for stateless design

An important observation we can make from these readings is that while the change in the time to download the layers consistent with our design assumptions, the value of the total image pull time does not vary by the same amount. There is a high variance in the total image pull times of the docker swarm, particularly the code after the image layers have been downloaded. Even while measuring the performance of the original docker code for pulling an image, we could observe that while running multiple iterations of downloading the same image, the time to pull the total image had a huge variance. While this is certainly not desired, our design changes mainly aim at improving the time to download the image layers on the client machine and is what we need to consider.

We can observe that the stateless design offers little improvement in performance, even when the re-use rate is as high as 100 percent. Performance analysis of the modified workflow, helped us determine the cause for such an observation. We do save time in the workflow by reducing the

| Function S | Layer download time | Percentage change | Total image pull time |
|---|---|---|---|
| Docker workflow | 1.399 s | - | 9.231 s |
| 0 percent re-use | 1.493 s | +6.71% | 10.183 s |
| 100 percent re-use | 1.248 s | -10.82% | 9.158 s |

| Function M | Layer download time | Percentage change | Total image pull time |
|---|---|---|---|
| Docker workflow | 1.958 s | - | 10.379 s |
| 0 percent re-use | 2.169 s | +10.78% | 11.321 s |
| 100 percent re-use | 01.538 s | -21.56% | 10.606 s |

| Function L | Layer download time | Percentage change | Total image pull time |
|---|---|---|---|
| Docker workflow | 02.126 s | - | 9.865 s |
| 0 percent re-use | 2.460 s | +15.70% | 11.401 s |
| 100 percent re-use | 1.766 s | -16.93% | 11.033 s |

Table 7.5: Table representing performance of function deployments for stateful design

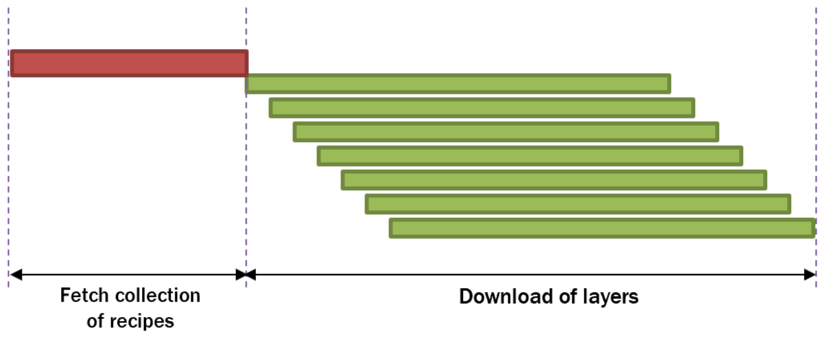


Figure 7.1: Activity diagram of stateless design

number of bytes transferred. However, before fetching the layers, we need to fetch the collection of recipes to determine the list of chunks that need to be sent over. This call is expensive, particularly taking around 700-1000 ms(depending on the size of the function). Intrinsic to the design, we need the collection of recipes before starting the download of the layers, and hence this part of the workflow cannot be parellelized with the download of the layers. The overhead of fetching the recipes hence cancels the benefits gained. Figure 7.1 describes the overhead of fetching the recipes diagrammatically.

In pursuit of a solution to get rid of the overhead motivated us to design the stateful approach. In the stateful design, since the registry is aware of the contents of the client stash, there is no overhead associated with pre-fetching these values in the critical pull workflow. We hence save precious time by offloading this task to an earlier point in time, having been carried out by the synchronization service.

## 7.6 Size and locality of client stash

Some important metrics corresponding to the sizes of the chunks and the generated keys are presented in table 7.6. We can see that 1GB of client stash could provide us $2^{21}$ blocks from deduplication. We would need to dedicate an optimal portion of our corpus of most popular chunks used.

| Parameter | Value |
|---|---|
| Size of chunk | 512 Bytes |
| MD5 fingerprint | 128 bits |
| Base 64 of fingerprint | 22 Bytes |
| Chunks in 1 GB DB | $2^{21}$ |
| Size of keys for 1 GB DB storage | 44 MB |

Table 7.6: Sizes of chunks and keys

For determining the size of the DB, the locality of the client stash is a significant factor to consider. The locality of DB can be chosen from a list of alternatives:

- In-memory database. If the server has 128 GB of memory and we reserve 10 percent of the capacity, this would be around 12.8 GB.
- On-disk DB. If the server has around 1 TB of disk space, and we reserve 10 percent of the capacity, this would be around 100 GB.
- External DB shared between a group of closely connected client machines. In data centers, we could have a database per rack. This DB can be around 1-2 TB in size.

As we move from an in-memory database to a remote external database, we can increase the size of the database and hence have a large number of chunks, thus achieving a larger deduplication ratio. However, while still faster than a remote server, retrieval from an external database would be relatively slower than an in-memory or an on-disk database. A comparative performance analysis implementing all the alternatives would help us arrive at a better conclusion to making a befitting choice.

# 8. CONCLUSION

## 8.1 Conclusion

In this thesis, we presented a novel method to reduce cold start latency using deduplication of data. To achieve this, we developed a prototype to demonstrate the reduction of transfer times during the deployment of a function, in a test setup built using OpenFaaS and Docker Swarm. We could demonstrate that with a high re-use rate, depending on the size of the function, we can reduce the latency of the transfer of data between 15 to 20 percent.

We also proposed a dynamic approach to design a global corpus of most popular blocks used for deduplication, and develop an encoding scheme to identify these blocks. We also explored several options which can be used to achieve a good rate of deduplication, and provided some metrics on the potential amount of deduplication we can achieve with our strategy. We also explored some metrics for a more practical deployment of our prototype and discuss some challenges for the scalability of our system.

## 8.2 Limitations

Our proposed solutions has some limitations we need to overcome. While platforms like OpenFaaS are popular as an open-source alternative, serverless platforms like AWS Lambda and Azure Functions, hold the state of the art implementation in this domain. We need to perform an analysis with real serverless computing samples, on the high-end data center infrastructure used in such commercial serverless platforms, to have more a more concrete estimate on deduplication ratio we can generally achieve. A low re-use rate, would in fact induce an overhead to slow down the existing latency metrics.

A major limitation of our stateful design approach, is that it hurts the ability of the registry service to chose from a wide range of machines for function placement. Instead, while using our approach, it is limited to deploy only to machines, whose updated stash state representation data it readily has access to. The engineering and synchronization efforts to develop a synchronization service which ensures that the state data of the client stash is synchronized with the registry service are significant.

## 8.3 Future Work

As described in section, 8.2, only an analysis with real computing samples and a high end infrastructure would give us an accurate estimate on the range of deduplication ratio we can achieve. We need to carry out such an analysis to demonstrate the working of our model in real-life scenarios.

In the current deduplication values we achieve, we are only able to incorporate similarity between chunks of files when the chunking offsets are aligned with the file lengths. There may be more similarity between byte streams of data within dissimilar files, which are not aligned with the chunks of 512 bytes currently observed. We need to explore further strategies for identifying such duplication.

Also, we need to perform a sensitivity test with various network parameters such as network latency and bandwidth, since our results may change with a variation in these values.

# REFERENCES

[1] "Documentation source for openfaas images." https://github.com/openfaas/faas/blob/master/docs.

[2] "Documentation license for openfaas." https://github.com/openfaas/faas/blob/master/LICENSE.

[3] "Github source for docker docs on swarm architecture." https://github.com/docker/docker.github.io/blob/master/engine/swarm/images.

[4] "Documentation license for docker." https://github.com/docker/docker.github.io/blob/master/LICENSE.

[5] "Github source for docker docs on container layers." https://github.com/docker/docker.github.io/blob/master/storage/storagedriver/images/container-layers.jpg.

[6] "Tar node segments." https://jackrabbit.apache.org/oak/docs/nodestore/segment/tar.html.

[7] "License for tar node segments." https://jackrabbit.apache.org/oak/docs/license.html.

[8] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[9] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *2008 Grid Computing Environments Workshop*, pp. 1–10, 2008.

[10] "Amazon Simple Storage Service(s3)." https://aws.amazon.com/s3/.

[11] "Amazon Simple Notification Service(sns)." https://aws.amazon.com/sns/.

[12] "Amazon Simple Queue Service(sqs)." https://aws.amazon.com/sqs/.

[13] "AWS Lambda." https://aws.amazon.com/lambda/.

[14] "Azure Functions." https://azure.microsoft.com/en-us/services/functions/.

[15] “Google Cloud Functions.” https://cloud.google.com/functions.

[16] “Apache openwhisk.” https://openwhisk.apache.org/.

[17] “OpenFaaS.” https://github.com/openfaas/faas.

[18] “Container Sandbox using gVisor.” https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime.

[19] “Micro VMs using AWS Firecracker.” https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.

[20] “Blog on cold start in AWS Lambda.” https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/.

[21] “Blog on cold start in Azure Functions.” https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/.

[22] “Blog on dealing with cold starts in AWS Lambda.” https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532.

[23] “Scaling on invocation of AWS Lambda.” https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html.

[24] “Asynchronous invocation.” https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html.

[25] “Blog on concurrent invocations.” https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f/.

[26] “Can We Solve Serverless Cold Starts?.” https://dashbird.io/blog/can-we-solve-serverless-cold-starts/.

[27] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, “Cold start influencing factors in function as a service,” 10 2018.

[28] D. T. Meyer and W. J. Bolosky, “A study of practical deduplication,” *ACM Trans. Storage*, vol. 7, Feb. 2012.

[29] A. El-Shimi, R. Kalach, A. Kumar, A. Oltean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (USA), p. 26, USENIX Association, 2012.

[30] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, (USA), p. 4, USENIX Association, 2012.

[31] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *ACM Trans. Storage*, vol. 8, Dec. 2012.

[32] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[33] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[34] "Funtion scaling on Azure platflorm." https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.

[35] "Preloading containers to reduce cold-start ." https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0.

[36] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, (Renton, WA), USENIX Association, July 2019.

[37] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, (USA), p. 923–935, USENIX Association, 2018.

[38] “AWS X-Ray.” https://aws.amazon.com/xray/.

[39] “Lambda and serverless is one of the worst forms of proprietary lock-in we’ve ever seen in the history of humanity.” https://www.theregister.com/2017/11/06/coreos_kubernetes_v_world/.

[40] “Serverless Open-Source Frameworks: OpenFaaS, Knative, More.” https://epsagon.com/blog/serverless-open-source-frameworks-openfaas-knative-more/.

[41] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Serverless computation with openlambda,” in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, (Denver, CO), USENIX Association, June 2016.

[42] “OpenFaas on private clouds.” https://www.openfaas.com/blog/ofc-private-cloud/.

[43] “OpenFaas on Github.” https://github.com/openfaas/faas.

[44] “Docker-swarm.” https://docs.docker.com/engine/swarm/.

[45] “Kubernetes.” https://kubernetes.io/.

[46] “OpenFaas Docs.” https://docs.openfaas.com/.

[47] “Docker Registry.” https://docs.docker.com/registry//.

[48] “Docker Hub.” https://hub.docker.com/.

[49] “GRPC.” https://grpc.io/docs/.

[50] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 {USENIX} Annual Technical Conference*, pp. 305–319, 2014.

[51] “Docker Documentation.” https://docs.docker.com.

[52] “Union mount filesystem.” https://lwn.net/Articles/324291/.

[53] “Tar file.” https://www.gnu.org/software/tar/manual/html_node/Standard.html.

[54] “npm rank.” https://gist.github.com/anvaka/8e8fa57c7ee1350e3491.

[55] R. L. Rivest, “The md5 message-digest algorithm,” RFC 1321, RFC Editor, April 1992. http://www.rfc-editor.org/rfc/rfc1321.txt.

[56] “Sha256.” https://worldwide.espacenet.com/patent/search/family/025175897/publication/US2002122554A1?q=pn%3DUS6829355.

[57] “Redis- an in-memory data structure store.” https://redis.io/.